O'REILLY®


图灵程序设计丛书



PWA开发实战

Building Progressive Web Apps

通过实践详解如何构建现代渐进式Web应用

[以] 塔勒·爱特尔 著

张俊达 译


中国工信出版集团


人民邮电出版社
POSTS & TELECOM PRESS

译者介绍

张俊达

前端开发工程师，关注前端领域的新技术，乐于分享。译作包括《React快速上手开发》《同构JavaScript应用开发》。

数字版权声明



图灵程序设计丛书

PWA开发实战

Building Progressive Web Apps

[以] 塔勒 • 爱特尔　著
张俊达　译

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®
O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北　　京

图书在版编目（CIP）数据

PWA开发实战 / （以）塔勒·爱特尔（Tal Ater）著 ；
张俊达译. -- 北京 : 人民邮电出版社, 2019.1
（图灵程序设计丛书）
ISBN 978-7-115-50200-1

Ⅰ. ①P… Ⅱ. ①塔… ②张… Ⅲ. ①网页制作工具
Ⅳ. ①TP393.092.2

中国版本图书馆CIP数据核字(2018)第265309号

内 容 提 要

本书通过实际操作帮助读者透彻地理解现代渐进式 Web 应用开发，指导读者学会利用原生应用的特性构建 Web 应用。主要内容包括：某酒店网站构建全流程，开发渐进式 Web 应用时一些需要重点考虑的因素，离线优先的 Web 应用的原则，渐进式 Web 应用为用户界面带来的一些新挑战和新机会，等等。

本书适合 Web 开发人员和业务管理人员阅读。

◆ 著　　[以] 塔勒·爱特尔
译　　张俊达
责任编辑　岳新欣
责任印制　周昇亮

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：13.75
字数：325千字　　2019年1月第1版
印数：1－3 000册　　2019年1月北京第1次印刷
著作权合同登记号　图字：01-2018-7358号

定价：79.00元
读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字 20170147 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过图书出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

献给我最爱的两位女士和她们的度假帽。

前言

渐进式 Web 应用（progressive Web app，PWA）是现代 Web 应用的一种激动人心的新形式。它利用了最新的 Web 功能，结合了原生移动应用的独特特性与 Web 的优点，为用户带来了新的体验。

本书将帮助你通过动手实践透彻地理解现代渐进式 Web 应用开发。

你将学习如何利用曾经专属于原生应用的特性构建 Web 应用。你将能够通过推送通知联系用户，在用户的主屏幕上占领先机，显著地加快网站速度，并且无论用户的网络连接状况如何，都能为其提供一个全功能的应用。

本书采用动手实践的学习方式，将一个现有网站逐章转化成现代渐进式 Web 应用。

读者对象

本书主要是为开发人员准备的。如果你想利用已有的 Web 开发技能，学习如何构建现代渐进式 Web 应用，那么本书就是为你而写的。

本书假定你至少对使用 HTML 和 JavaScript 进行 Web 开发有基本的了解，但不要求你熟悉 JavaScript 相对较新的特性，比如 ECMAScript 2015、promise，以及 ECMAScript 2017 的异步函数。如果你已经了解这些现代语言结构，完全可以跳过或快速浏览讲解这些内容的注释。

本书可以帮助非技术人员熟悉并基本了解现代渐进式 Web 应用的功能。其中很多章都包含了案例研究，这些案例是通过对世界上最有影响力的一些网站背后的团队进行访谈得到的，包括 Twitter、《华盛顿邮报》、Housing.com 和 Lyft。无论你是管理人员、设计师、产品经理，还是任何参与原生或 Web 应用决策的其他人员，了解如今可能实现的技术，将有助于你更有效地开展工作。

本书内容

哥谭帝国酒店是本书虚构的酒店，它的网站很简单。阅读本书时，你将接管这个网站，通

过 service worker 技术对它进行增强，使其几乎可以瞬间加载（哪怕是在最慢的连接状况下），并确保所有的功能都可以在用户完全离线时使用（包括查看用户的预订，甚至是发起新预订）。你将学习如何让用户添加一个图标到手机的主屏幕上，并从此处启动你的渐进式 Web 应用。最后，为了实现原生应用般的体验，你将添加推送通知，如此一来，就算用户离开你的网站，你也能联系并重新召回他们。

本书还探讨了开发渐进式 Web 应用时需要重点考虑的一些因素，并专注于帮助你切实理解这些概念，进而成为更高效的开发人员。此外，本书还介绍了实用的开发工具、安全因素，以及 service worker 的生命周期。

虽然本书大部分章节都集中在通过动手实践进行学习上，但其中两章（第 5 章和第 11 章）会让你思考渐进式 Web 应用提供的新功能，让你意识到它们不仅仅是应用于你的应用的一套新技巧。

第 5 章探讨了离线优先的 Web 应用原则，以这种方式构建的现代 Web 应用不会把失去网络连接视为错误，而是能够为之做好计划并优雅地处理。

第 11 章探讨了渐进式 Web 应用为用户界面带来的一些新挑战和新机会。渐进式 Web 应用改变了用户对 Web 应用的期望。其中一些挑战包括：增强用户的信任，让他们相信离线时自己的数据不会丢失；告诉用户离线时看到的内容可能是几小时之前的，并且让用户知道无论发生了什么重要的变化，应用会发送通知给他。若应对得当，这些挑战也是大好时机，可以借此增强用户对应用的信任，提高转化率，并在用户的手机上取得永久性位置。

本书最后介绍了一些即将到来的技术和浏览器 API，它们将进一步推动渐进式 Web 应用的发展。

排版约定

本书采用以下排版约定。

- **黑体**
  表示新术语或者重点强调的内容。
- 等宽字体（`constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。
- 等宽粗体（**`constant width bold`**）
  表示应该由用户输入的命令或其他文本。
- 等宽斜体（*`constant width italic`*）
  表示应该由用户替换或取决于上下文的值。



该图标表示渐进式 Web 应用的案例研究。



该图标表示一般说明。



该图标表示从另一个角度看待同一问题。



该图标表示警告或提醒。

代码示例

补充材料（包括示例代码、练习题等）可以从 https://github.com/TalAter/gotham_imperial_hotel 下载。

本书旨在帮助你做好工作。一般来说，你可以在程序和文档中使用本书的代码。除非你使用了很大一部分代码，否则无须联系我们获得许可。例如，使用本书的几段代码编写一个程序不需要获得许可，销售和分发 O'Reilly 书中示例的光盘则需要获得许可。引用书中示例代码来回答问题无须获得许可，把书中的大量示例代码放入你的产品文档则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。例如："*Building Progressive Web Apps* by Tal Ater (O'Reilly). Copyright 2017 Tal Ater, 978-149-196165-0"。

如果你觉得自己对示例代码的使用超出了合理引用或者上述许可的范围，请随时通过 permissions@oreilly.com 联系我们。

O'Reilly Safari



Safari（之前称作 Safari Books Online）一个针对企业、政府、教育者和个人的会员制培训和参考平台。

会员可以访问来自 250 多家出版商的上千种图书、培训视频、学习路径、互动式教程和精选播放列表，这些出版商包括 O'Reilly Media、Harvard Business Review、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Adobe、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 等。

要了解更多信息，可以访问 http://www.oreilly.com/safari。

联系我们

请把对本书的评价和问题发给出版社。

美国：

> O'Reilly Media, Inc.
> 1005 Gravenstein Highway North
> Sebastopol, CA 95472

中国：

> 北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
> 奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属页面，你可以在那里找到本书的相关信息，包括勘误表、示例以及其他信息。[1] 本书的网页地址是：http://shop.oreilly.com/product/0636920052067.do

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

致谢

我首先要感谢 Alex Russell 和 Jake Archibald。在 2014 年的谷歌 I/O 开发者大会上，我偶然听到了他们的演讲。在那之前，我甚至不知道本书中涵盖的任何内容意味着什么。他们的演讲和本书的写作帮助我增加了了解。

我还要感谢在现实世界中构建渐进式 Web 应用的优秀人才，特别是那些在本书中分享经验的人。感谢 Joey Marburger、Ritesh Kumar、Rahul Yadav、Nicolas Gallagher、Chris Nguyen 和 Jeremy Toeman。

感谢 O'Reilly 团队帮助我出版了本书，包括 Ally MacDonald、Jeff Bleiel、Sonia Saruba、Colleen Cole、David Futato、Rebecca Demarest、Heather Scherer、Ellen Troutman、Amanda Kersey，以及 Karen Montgomery 和她的戴胜鸟（本书封面上的动物）。

还要感谢谷歌、Opera、Mozilla 和微软的开发团队，在他们的帮助下，本书和本书中介绍的技术才成为了现实。感谢优秀的开发者们帮助我更好地理解了这项技术。感谢 Jeffrey Posnick、Addy Osmani、Matt Gaunt 和 Paul Kinlan。

注 1：本书中文版的勘误请到 http://www.ituring.com.cn/book/2040 查看和提交。——编者注

感谢 Alex Feyerke 和 Hoodie 团队，他们围绕离线优先做出了开创性的工作，并通过写作记录下来。本书中的不少内容都是受到了他们的启发。

编写这样一本书最困难的一点，是在一年多的时间里几乎没有收到外部反馈，这带来了很大的不确定性。我要感谢所有在本书写作过程中抽出时间向我提供反馈意见的人。感谢 James Stanley、Patrick Conant、Fabio Rotondo、Neville Franks 和 Florian Semrau。

最后，我要感谢本书的技术审阅者，他们对本书进行了全面的技术审查，以保障我所写的内容确实有意义。感谢 Andreas Bovens、Kenneth Rohde Christiansen、Patrick Kettner 和 Thomas Steiner。

电子书

扫描如下二维码，即可购买本书电子版。



第1章

渐进式Web应用介绍

> 词是被遗忘的名之浅影。因为名有力量，所以词也有力量。词可以点燃人们心中的火焰，也可以让最狠的心流下眼泪。七个词便可以让一个人爱上你，十个词便可以让最坚强的人顿失意志。但是词不过是火之描绘，而名才是火之本身。
>
> ——Patrick Rothfuss，《风之名》

每隔几年，Web 就会经历一个关键性时刻。在这个时刻，几种独立的技术相互碰撞，在公众中引起轰动。这些技术可能已经诞生多年，也可能是刚刚获得浏览器支持的新技术。但对于旁观者来说，似乎在一个闪光的瞬间，Web 突然向前跃进了一步。

Ajax 技术就是这样。它似乎是某一天不知从何处突然蹦出来的（尽管很多底层技术已经存在多年，如 XMLHttpRequest），改变了我们对 Web 的理解：一系列相互链接的、大部分是静态的页面。

Ajax 本身只是 Web 2.0 革命的一部分，而 Web 2.0 是另一个强大的名字，它在 2004 年从天而降，一夜之间引爆了世界。

几年后，**移动优先**（mobile-first）出现了，它标志着我们对 Web 开发的看法发生了转变。通过给一套设计原则命名，这两个单词获得了难以置信的力量。只用这两个单词，我们就可以大声说，用户坐在台式机前，用 20 英寸的显示器和一根连接到墙上的电缆上网的时代已经结束了。这两个单词让我们明白，是时候改变 Web 开发的方式了。

这样的时刻往往不是在技术诞生的时候出现的，而是在它被命名的时候。

名字就是有这样的力量：它让我们掌握新思想、讨论新概念，提醒我们注意在表面下酝酿的风暴。

一个同样巨大的转变正在发生。幸运的是，它有一个名字。[1]

1.1 Web反击战

渐进式 Web 应用是一种崭新的 Web 应用，它结合了原生应用的优点和 Web 少冲突的特点。

渐进式 Web 应用始于简单的网站，但随着用户的使用，它不断获得新的权限，并从网站变成一种更像传统原生应用的形式。

想象一下，你早上醒来，抓起手机，浏览本地列车公司的网站。你快速查看了上班需要乘坐的列车的时间表，然后关闭浏览器，并把手机放进口袋。下班时，你再次访问该网站，查看下一趟列车何时发车（你甚至没有注意到正在乘坐的电梯没有手机信号，因为列车公司的网站依然在运行，即使你处于离线状态）。次日，当你再次访问该网站时，浏览器询问你是否要添加快捷方式到主屏幕上，你欣然同意。当天晚些时候，当你从主屏幕的图标登录该网站时，它通知你，由于施工，列车可能会延误，并问你是否想接收关于列车时刻变化的后续通知。第三天早上，当你醒来时，手机收到消息推送，说该列车会延误 15 分钟。你按下闹钟上的延时按钮，又多睡了一会。

渐进式 Web 应用从一个简单的网站开始，慢慢获得新的权限，直到和原生应用一样，而不是试图把你送到应用商店，希望你安装应用。就这样，列车公司在你的手机上一步一步赢得了永久性的位置。

这种新的渐进增强模型，取代了只提供“安装”和“不安装”两种选择的原生应用。渐进式 Web 应用能赢得用户的信任，并按需获得新的权限。

你可能会问，为什么这是对原生应用的改进呢？我们为什么不坚持使用原生应用呢？好吧，除非你是少数的幸运儿，否则你应该知道，原生应用已经行不通了。用户安装应用的可能性在逐年减小，获取新用户的成本在逐渐增长，留住用户变得越来越困难。

1.2 当前的移动领域

当第一款 iPhone 在 2007 年推出时，它的杀手级功能是让你能够在手机上浏览网站。当移动应用于一年后诞生时，开发人员终于能够突破网页的功能限制了（同时，由于应用商店的引入，也面临许多新的限制）。

Web 具有高级图像技术、地理位置识别、消息推送、离线可用性、主屏幕图标等特性，但在许多开发人员的眼中，Web 似乎相形见绌。原生应用接管了全球（和我们的手机）。

但这种趋势正在转变。虽然我们在手机和移动应用上花费的时间比以往任何时候都多，但我们使用的应用却越来越少。用户安装的应用少了，而且只使用其中几个。如果你的应用在应用商店中排在前 10 位，你可能不会有这种困扰。但是，现在尝试将一款新的应用打

注 1：更确切地说，幸运的是，Alex Russel 和 Frances Berriman 有一天共进晚餐，并想出了一个名字。详细内容请参见 Alex Russel 的博客文章：“Progressive Web Apps: Escaping Tabs Without Losing Our Soul”。

入市场几乎是不可能的，成本就更别提了。



移动用户的行为

根据 2016 年 comScore 的报告，在移动设备上，平均每人将 84% 的时间花在最流行的 5 个应用上。抱歉，这些不是你的应用。在平板电脑上，这个比例甚至更高，用户将 95% 的时间花在了这 5 个应用上。

该报告还表明，移动网站比原生应用更容易获得大量用户。拥有 500 万以上访问者的移动网站接近 600 个，比拥有类似受众的原生应用多 4.5 倍。排名前 1000 位的移动网站拥有的用户数量几乎是排名前 1000 位的原生应用的 3 倍，而且前者的用户增长速度是后者的两倍。

让用户安装并使用应用，如同在一个漏斗形空间里挣扎求生。用户需要知道你的应用（通过传统的在线广告或你的网站），然后必须访问其在应用商店中的页面，接下来需要点击安装。他们需要同意授予应用不同的权限，然后等待应用下载并安装。最后，他们至少要启动一次应用，甚至使用它。

当用户安装他们知道并喜欢的应用（比如 Twitter 或者 Facebook）时，这个漏斗看起来并不算太糟糕。但是大量研究表明，在这个漏斗的每一个环节，平均有 20% 的用户丢失了。应用开发人员为广告点击付费后，发现只有不到 20% 的用户实际启动了应用，这种情况并不罕见。

网站竭尽所能地让你安装他们的应用，甚至采用了一种新的广告方式。你肯定见过这种广告：当你打开一个网站，想看一篇短文或者查询明天的天气时，发现所需信息近在眼前，但是一个横幅广告随即弹了出来，挡住了你想看的内容。它问你是否愿意安装一个应用，而不直接阅读已经在你眼前的内容。

有些人称之为**全页间隙式广告**（full-page interstitial ad）。我喜欢一个较短的名字：**用力关门**（the door slam，如图 1-1 所示）。要详细了解全页间隙式广告的作用与副作用，请参见附录 B。

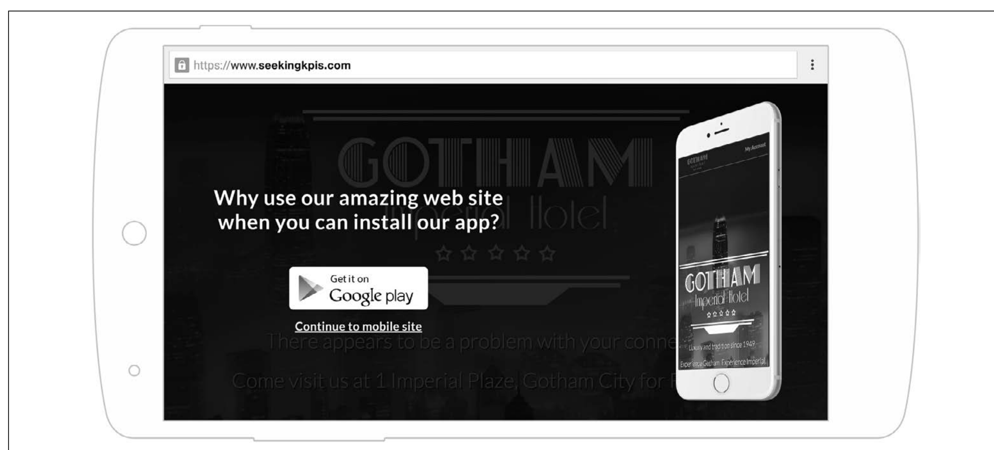


图 1-1：一种常见的“用力关门”式广告

让用户在手机上安装你的应用是一场残酷、昂贵的战斗。然而，原生应用相较于 Web 的优势，让应用开发者心甘情愿为每一次应用安装忍受痛苦。

与传统网站不同，原生应用的生命并不限于用户首次发现它到离开这段短暂的时间（有时候，这段时间只有几秒钟）。一旦安装，原生应用就在你的主屏幕上占据了永久性的位置。它可以随时通过消息推送提醒你它的存在。这让开发者有很多机会在应用的长生命周期里尝试获得投资回报。

但随着渐进式 Web 应用的推出，潮流终于开始转向了。这些超能力所带来的优势曾经是原生应用专有的，但现在可以移植到 Web 应用上了。将这些能力与 Web 少冲突、一步漏斗的特点（使用点击链接取代安装应用）结合起来，你就会明白为什么用户、Web 开发者和企业都能因拥抱渐进式 Web 应用而获益良多。

1.3 渐进式Web应用的优势

随着上述超能力的引入，渐进式 Web 应用实现了我们寄予原生应用的很多期望。

以下是本书中将介绍的部分优势。

无连接状态下的可用性

和传统网站不同，渐进式 Web 应用不依赖于用户的连接状态。当用户访问一个渐进式 Web 应用时，它会注册一个 service worker（参见 1.4 节），service worker 可以检测并响应用户连接状态的变化。无论用户是离线、在线，还是处于网络不稳定状态，它都可以提供完整的用户体验。

用户可以在穿越大西洋的航班上使用你的渐进式 Web 应用，甚至可以进行操作（例如发布信息、回复事件或者评论文章），用户知道他的操作会在网络恢复之后立刻完成，即便他关闭了应用和浏览器。详情请参见第 7 章。

渐进式 Web 应用引入了一定程度的可靠性，将用户的信任程度提升到原本只有原生应用才能达到的程度。用户知道他可以在任何时间打开 WhatsApp 应用，写一条短信，然后关掉手机，而无须担心连接状态。到目前为止，Web 网站还没获得这种信任，这也是用户更偏向于使用原生应用的原因之一。

加载速度快

使用 service worker，我们可以创建一个瞬间运行的网站，无论用户的网速极快，还是使用不可靠的 2G 连接，甚至是在完全没有网络连接的状态下。网站可以在几毫秒内加载，这比过去的 Web 要快得多，甚至比原生应用还要快。在第 5 章中，我们将学习如何实现这一点，并探讨离线优先的原则。

推送通知

渐进式 Web 应用可以向用户推送通知（甚至是在用户离开网站几天后）。这些通知提供了一个好机会，让你得以重新吸引用户，并提醒他们回到应用。渐进式 Web 应用的通知是完全原生化的，和原生应用的通知没有区别。参见第 10 章了解更多关于推送通知的内容。

主屏幕快捷方式

一旦用户表现出对渐进式 Web 应用感兴趣，浏览器就会自动建议用户添加快捷方式到主屏幕上——和原生应用完全一致（如图 1-2 所示）。参见第 9 章，了解如何在用户的主屏幕上占据位置。

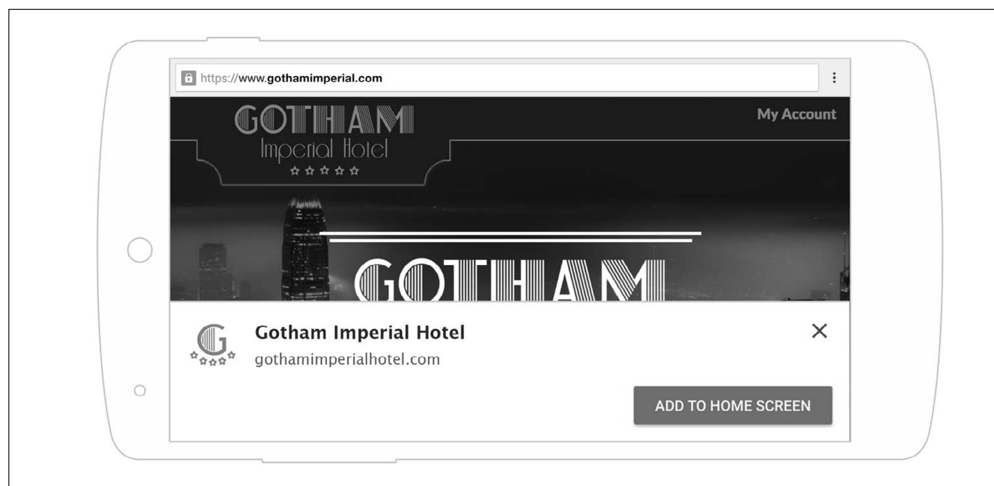


图 1-2：应用安装横条

媲美原生

渐进式 Web应用从主屏幕启动的过程可以完全原生化，和原生应用相似。前者在加载过程中可以显示启动画面；可以以全屏模式运行，摆脱浏览器和手机系统的 UI 界面；甚至可以锁定屏幕方向（这对于游戏应用而言至关重要）。

参见第 9 章了解更多细节。



Lyft——平台越多，乘客越多

除了用户体验方面的好处之外，渐进式 Web 应用也为采用它的企业提供了额外的好处。

Lyft 提供流行的打车服务，这家公司的收入完全依赖移动应用。

在不断努力获得更多用户的过程中，Lyft 公司发现自己不得不支持越来越多的设备和移动端操作系统版本。随着应用的演进，Lyft 不得不放弃支持旧版本的 iOS 和安卓系统，否则维护成本将不断增长。Lyft 没有放弃这些潜在的客户（占据了大约 8% 的 iOS 用户和 3% 的安卓用户），而是构建了一个渐进式 Web 应用。

采用了渐进式 Web 应用之后，Lyft 团队成功减少了支持多应用和多设备的技术和运营成本。更重要的是，他们不仅获取到了使用 iOS 和安卓系统的新用户，还兼顾了他们以前忽略的 Windows Mobile 和 Amazon Fire 用户。

1.4 浏览器标签页、Web和service worker

任何渐进式 Web 应用的核心都是 service worker 技术。

service worker 体现了我们对 Web 开发看法的转变。我们花几分钟了解一下这项技术的定位，这对于理解它的潜力至关重要。

在 service worker 出现之前，我们的代码只能在服务器或者浏览器端运行，而 service worker 的出现引入了另外一层。

service worker 是一种脚本，可以通过注册它来控制你站点中的一个或多个页面。一旦安装完毕，service worker 就会独立存在，而不是属于某个浏览器窗口或者标签页。

service worker 可以监听并响应在其控制之下的所有页面的事件。向 Web 请求文件等事件，可以被它拦截、修改、传递并返回给页面（参见图 1-3）。

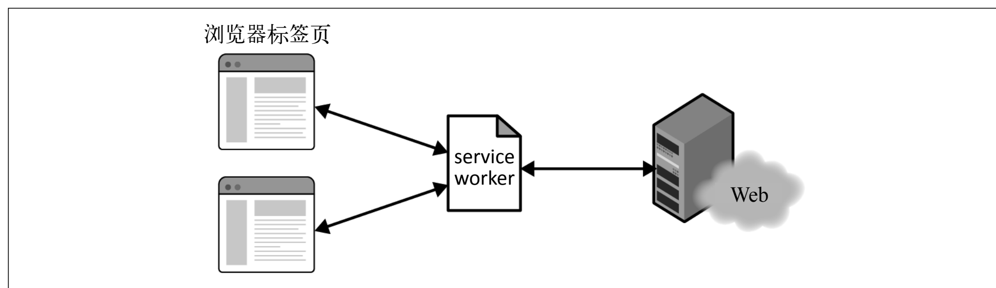


图 1-3：浏览器标签页、Web 和 service worker 的关系

这意味着页面和 Web 之间增加了一层，它可以响应请求，而无论网络连接状态如何。service worker 层甚至可以在用户离线的情况下正常工作。它可以检测到离线状态或者服务器响应慢的情况，并返回缓存内容取而代之（参见图 1-4）。

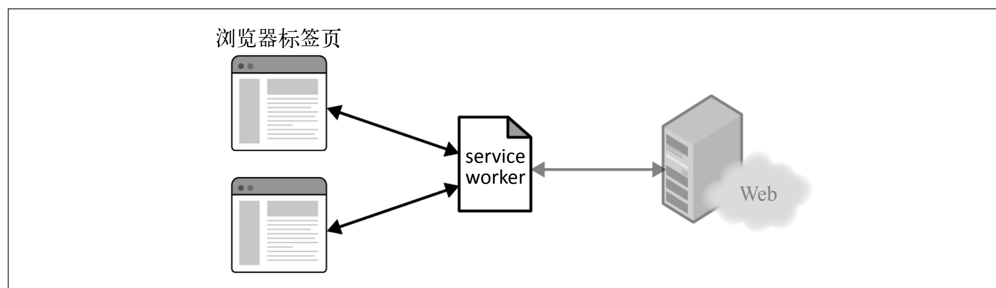


图 1-4：用户离线时，页面与 service worker 进行通信

再进一步，这意味着即使用户关闭浏览器中运行着你的应用的所有标签，service worker 层依然可以和服务器通信（参见图 1-5）。它可以接收并显示推送通知，或者确保任何用户操作都能够被传递到服务器（即使用户一边走进电梯一边进行操作，并在重新建立网络连接前关闭了应用）。

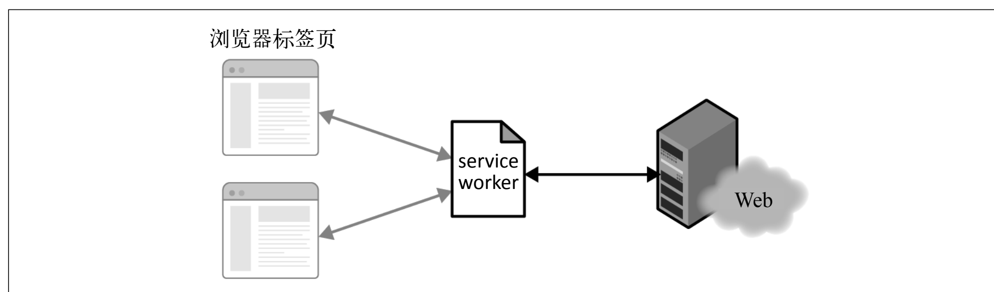


图 1-5：在用户离开页面后，service worker 与服务器进行通信

现在你知道为什么 service worker 是每一款渐进式 Web 应用的核心了。它的持久性让渐进式 Web 应用能够实现我们对于一款应用的期望。它弥补了 Web 应用的缺失环节，在过去只有原生应用能够做到的事情，现在渐进式 Web 应用也能做到了。

但也许 service worker 最大的优势在于它就是简简单单的 JavaScript 文件。它的写法和其他 JavaScript 代码一样，前端开发者没有额外的语言学习成本。

对于开发者来说，若能理解 service worker 和本书所涉及的其他相关技术，收益将是巨大的。这些技术让我们可以利用现有的 Web 技术，包括 JavaScript、HTML 和 CSS，编写出完全可以媲美甚至超越原生移动应用的 Web 应用。

第 2 章

你的第一个service worker

2.1 设置示例项目

本书将通过动手实践的方式学习渐进式 Web 应用。

从本章开始，我们将接管一个简单的 Web 应用（属于本书虚构的哥谭帝国酒店），并逐章对其进行改善。每一章都会在前一章的基础上进行提升，并且在每章的最后，你都可以得到一个可用的 Web 应用。

到本书结束时，你将把这个简单的网站变成一个功能全面的渐进式 Web 应用。

为了按照代码示例进行学习并亲自动手尝试，你可以把应用的源代码复制到你的本地机器上。你可以在哥谭帝国酒店的 GitHub 仓库（https://github.com/TalAter/gotham_imperial_hotel/issues）中获取源代码。

请注意，为了复制代码并在本地运行，你需要能够在本地机器中运行 Git、Node.js 和 NPM。否则，你就需要通过其他的方式进行本书的学习（例如从 GitHub 直接下载源代码，并在远程服务器中运行），我并不推荐这样做。

首先，打开你电脑中的命令提示符（控制台），切换到你希望下载源代码的目录，并运行以下命令：

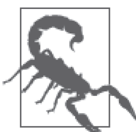
```
git clone -b ch02-start git@github.com:TalAter/gotham_imperial_hotel.git
cd gotham_imperial_hotel
npm install
```

上述命令会复制哥谭帝国酒店的 Web 应用的源代码，把分支切换到 `ch02-start`，并安装运行代码所需要的依赖库。

接下来，你可以使用下列命令来启动一个本地服务器，用浏览器打开你的站点：

```
npm start
```

现在如果你在浏览器中打开 http://localhost:8443/，应该能够看到哥谭帝国酒店的 Web 应用了。



如果浏览器中没有成功加载 Web 应用，请确认以下内容。

- 你已经安装 Git、Node.js 和 NPM，并且可以在命令行（例如 macOS 中的 Terminal 或者 iTerm，Windows 的命令提示符或者 Cygwin）中使用它们。
- 你已经遵循了前面所有的步骤。

如果你现在依然不能运行应用，请随时到我们的 GitHub 问题跟踪中寻求帮助。

现在，你可以在自己喜欢的 IDE 或者编辑器中打开该项目，并跟随本书将该站点转换成一个渐进式 Web 应用。

由于每一章的代码都建立在前几章所做的更改之上，所以在每一章开始前，你的代码都需要包含之前所有的更改。如果你打算跳过本书中的任何编码练习，或者整章跳过，则可以在命令行中运行以下两个命令，将代码切换到每章开头的状态：

```
git reset --hard
git checkout ch04-start
```

上述命令会重置所有本地已完成的修改，并把分支切换到那一章开始之前的状态。请确保将第二条命令中的分支名称更改成当前所在章节的名称。例如，当你开始阅读第 6 章的时候，只需运行 `git checkout ch06-start`，就可以切换到包含前面五章完成的所有更改的分支中。

2.2 欢迎来到哥谭帝国酒店

本书将通过虚拟的哥谭帝国酒店的网站这一项目来探索渐进式 Web 应用。

这个简单的网站包含两部分：

(1) 首页，包含了酒店介绍、地图、最近活动列表，以及一个发起新预订的表单；
(2) 个人账号页面，包含了用户的预订列表、最近活动，以及一个发起新预订的表单。

虽然看起来简单，但是这两个页面已经包含了构成重内容网站以及类似于原生应用的 Web 应用的大部分元素。

通过学习本书，你就可以把这个简单的网站变成一个功能齐全的渐进式 Web 应用。



挑战不同，实现方式各异

在探索渐进式 Web 应用的不同特性时，我们偶尔会从哥谭帝国酒店应用中后退一步，到一个不同的场景中探索相同的想法。

虽然我们的酒店应用更类似于一个传统的商务网站，但这些注解将从一个更类似于传统原生应用的应用的角度探索相似的挑战。通过探索这些方法的异同，我们可以更好地理解每项特性是如何适应到不同的项目中的，以及不同的企业如何从每项新特性中获益。

msger 是我们将在这些注解中探索的一个虚构的消息应用，这款应用允许用户发送 140 个字符的短消息，并能展示最新的用户消息流。当新消息出现的时候，它会被添加到消息流的顶部，旧消息将移到列表下方（见图 2-1）。

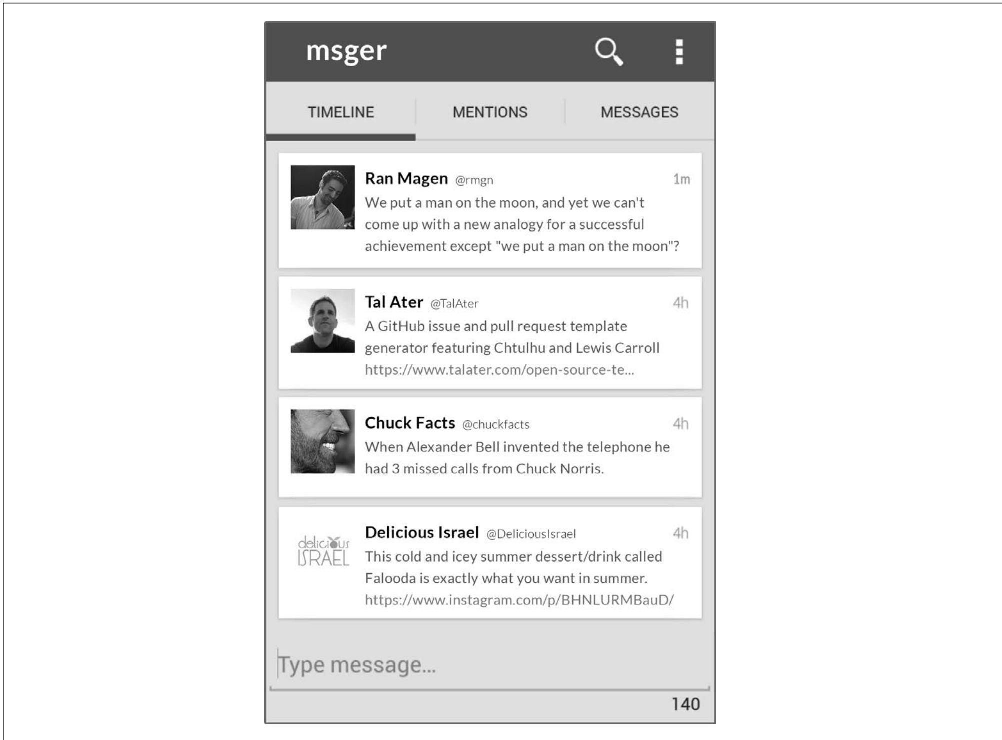


图 2-1：我们的示例消息应用

2.3 熟悉代码

在开始之前，我们先来熟悉一下这个应用的基本代码结构。

项目主目录中包含了两个最重要的目录。

public

包含了网站的所有客户端代码，以及运行代码所需的其他文件，例如图片和样式表。

server

包含了运行网站、跟踪预订、发送通知等的服务端代码。

本书中的所有编码练习只涉及 public 目录，但是你可能需要时不时关注一下 server 目录，特别是在第 10 章中。



写在介绍代码之前

如果你看看应用代码的初始状态，会发现这份代码非常简单。为了增强可读性以及清楚地展示我们将学习的关键原理，代码中有不少地方没有遵循最佳实践，甚至违背了常识。

当你学完本书，将有机会大幅改进这份代码。希望你不仅学会了如何从头开始构建一个渐进式 Web 应用，还学会了如何改进现有的项目，将其转化为一个渐进式 Web 应用。

本书中没有使用很多的现代 ES2015 语言结构，这样你就可以专注在本书的主题上，而不是那些你可能熟悉也可能陌生的新语法。要查看本书中的代码如何从 ES2015 中受益，请参阅附录 A。

2.4　当前的离线体验

读完上一节后，你应该已经拥有一份哥谭帝国酒店 Web 应用的代码副本，以及一个可运行该应用的本地 Web 服务器。

要确保你当前的代码是处于本章开始时的状态，可以在命令行中输入下列命令：

```
git reset --hard
git checkout ch02-start
```

接下来运行 `npm start` 命令，开启一个运行这个网站的本地 Web 服务器，然后在浏览器中打开它（http://localhost:8443/），你应该就能够看到这个网站的全貌了（如图 2-2 所示）。



图 2-2：哥谭帝国酒店首页

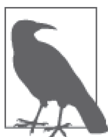
如今的 Web 就如同这个网站，内容丰富、界面美观、功能实用。但事实上，这样的 Web

是你从开发者的角度看到的。作为开发者，我们经常会通过比较先进的台式机、笔记本电脑或者移动设备访问网站。我们有着可靠的连接，或者是连接到本地服务器，或者是连接到一个距离很近的开发服务器。但是，用户体验到的 Web 应用可能是完全不同的。试想，当用户在离线的时候访问我们的网站会如何呢（见图 2-3）？



图 2-3：一位用户在电梯中访问了我们的示例 Web 应用

不幸的是，对于很多用户来说，这才是他们如今使用的 Web。终于，service worker 的出现让我们有机会处理这种情况了。



模拟离线状态

在本书使用这个示例应用的过程中，我们经常需要模拟离线状态。由于离线状态的本质是用户无法访问你的服务器，因此，模拟这种状态的一种方法是关闭你的开发服务器。

在运行着本地服务器的命令行中，按下 Ctrl+C 可以停止服务器进程。接下来，在浏览器中重新加载应用，就可以看到用户离线访问的效果了。

当你想要重新“上线”的时候，只需再次运行 `npm start` 命令即可。

这种基本的方法在开发过程中可以很好地模拟离线状态，但是一旦将代码发布到生产环境，每当你想进行某项测试时就“下线”生产服务器，这是行不通的。所幸，大部分现代浏览器都包含了用于模拟离线状态的工具，甚至可以模拟不同的连接速度（见图 2-4）。详情请参见 4.8 节。

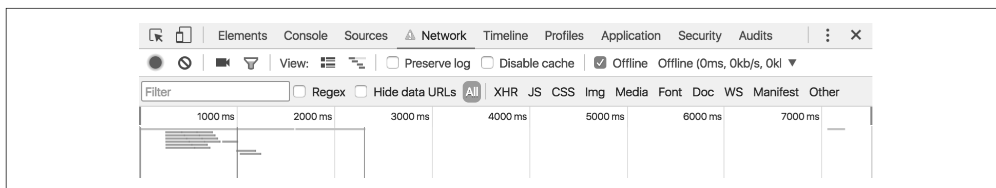


图 2-4：在 Google Chrome 中模拟离线状态

2.5 创建你的第一个service worker

让我们来控制用户的离线体验。

首先从当前页注册一个新的 service worker。打开 js/app.js 文件，在文件开头添加如下代码：

```
if ("serviceWorker" in navigator) {
  navigator.serviceWorker.register("/serviceworker.js")
    .then(function(registration) {
      console.log("Service Worker registered with scope:", registration.scope);
    }).catch(function(err) {
      console.log("Service worker registration failed:", err);
    });
}
```

代码首先验证了当前浏览器是否支持 service worker。然后通过调用 `navigator.serviceWorker.register` 方法注册了我们的 service worker，这个方法接收两个参数，第一个是我们的 service worker 脚本 URL，第二个是可选的选项对象（这里省略了这个参数，但会在 2.11 节探讨这个问题）。

通过在使用之前先测试 service worker 的支持情况，可确保不会将使用旧浏览器来访问应用的用户排除在外，并同时向使用现代浏览器的用户提供增强的体验。这种**渐进增强**式的实践是构建这个应用的核心（参见 2.6 节）。

上述的 `register` 调用会返回一个 promise。如果 promise 完成，就意味着 service worker 成功注册了，在 `then` 语句中定义的函数就会被调用。否则，如果 promise 遇到任何问题，就会执行 `catch` 块内定义的函数。

现在，如果你在浏览器中刷新示例应用，应该会在浏览器的控制台中看到一条错误信息，告诉你“Service worker registration failed.”。[1]

service worker 注册失败了，promise 的状态也因此失败，因为目前我们还没有创建 serviceworker.js 文件。

创建一个空文件，命名为 serviceworker.js，并放置在项目根目录的 public 文件夹中，即 public/serviceworker.js。此时，如果你刷新浏览器，应该会看到一条消息，告诉你“Service worker registered with scope: http://localhost:8443/”。虽然现在我们的 service worker 只是一个空文件，但它仍然是一个有效的 service worker，并且已经注册成功了。



可能你会想把 serviceworker.js 文件移动到项目的 js 子目录。请暂时把它放在根目录中。在 2.11 节中，你将会了解到这一点的重要性。

让我们开始探索 service worker 可以做的事情。

在 serviceworker.js 文件中添加下列代码：

注 1：如果你没有看到这条错误信息，请确保你的本地服务器正在运行，并阅读关于浏览器支持的内容（即 2.5 节中的“service worker 的浏览器支持度”。

```
self.addEventListener("fetch", function(event) {
  console.log("Fetch request for:", event.request.url);
});
```

这段代码通过调用 `self` 变量的 `addEventListener` 方法（在 service worker 中，`self` 指向 service worker 本身），在我们的 service worker 中添加了一个事件监听器。这个监听器会监听所有经过 service worker 的 fetch 事件，并运行我们接下来定义的函数，将事件对象 event 作为唯一参数传递。在我们定义的函数中，通过访问事件的 request 对象（这是 fetch 事件中的一个属性），把这次请求的 URL 打印到控制台中。

现在刷新页面，应该能看到页面发起的所有请求都被记录在浏览器的控制台中。（如果你没有在控制台看到任何 URL，可能是因为旧版本的空 service worker 依然在控制页面。参见“service worker 生命周期”获取相关技巧。）

service worker 生命周期

你可能已经注意到，当你修改 service worker 文件的时候，这些修改并没有在刷新浏览器之后立即生效。这是因为旧版本的 service worker 依然处于**激活**（active）状态，与此同时，新的 service worker 仍然处于**等待**（waiting）状态，直到旧版本不再控制页面为止。

虽然这看起来可能非常不方便，但它实际上是 service worker 的一项非常强大的特性。我们将在第 4 章更详细地探讨这一点。

为了简化开发，你可以告诉浏览器让新的 service worker 立即控制页面。在 Chrome 浏览器中，这可以通过开发者工具的 Application 选项卡实现：在“Service Workers”选项中，勾选“Upload on reload”（见图 2-5）。这样可以确保每次修改 service worker 并刷新页面时，新的 service worker 会立即控制页面。

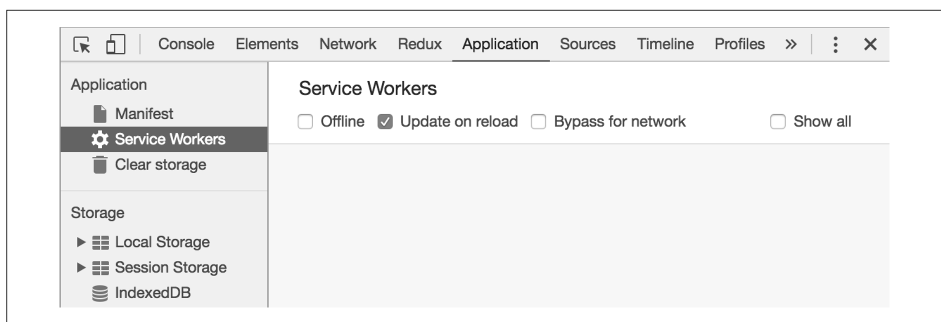


图 2-5：打开“Upload on reload”

上述的演示可能没有给你留下深刻的印象，那么请思考：我们的页面发起的每一个请求（包括向第三方服务器发起的请求）现在都会经过我们的 service worker。现在所有的这些请求都可以被拦截、分析，甚至被操控。

让我们通过一个例子看看这个特性有多强大。

把 serviceworker.js 中的代码替换成下列代码，并刷新你的浏览器：

```
self.addEventListener("fetch", function(event) {
  if (event.request.url.includes("bootstrap.min.css")) {
    event.respondWith(
      new Response(
        ".hotel-slogan {background: green!important;} nav {display:none}",
        { headers: { "Content-Type": "text/css" }}
      )
    );
  }
});
```

上述代码监听 fetch 事件，并检查每个请求的 URL 是否包含 `bootstrap.min.css` 字符串。如果包含，service worker 就会动态创建一个 `Response` 对象，其中包含了自定义的 CSS，并使用这个对象作为响应，而不会向远程服务器请求这个文件（见图 2-6）。



图 2-6：重写 CSS 请求，并修改背景颜色

仅仅用了短短几行 JavaScript 代码，我们就创建了一个可以拦截第三方服务器请求的 service worker，凭空编写了一个新的响应并展示给浏览器，看起来就像是服务端返回了结果一样。从本质上说，我们在浏览器内部创建了一个代理服务器。



service worker 浏览器的支持度

虽然 service worker 的规范在 2014 年才发布，但是浏览器接纳 service worker 的速度却出人意料地快。到 2015 年底，Chrome、Opera、Firefox 和 Samsung Internet 都已经支持 service worker 了。

在本书出版的时候，WebKit 团队正致力于将 service worker 融入到 iPhone 和所有基于 Safari 的浏览器上，另外 Microsoft Edge 团队也在努力当中。

要了解 service worker 的浏览器支持的最新状态，以及相关技术，请参见 Jake Archibald 的网页“Is ServiceWorker Ready?”。

2.6　什么是渐进增强

渐进增强（progressive enhancement）是我们的应用以及任何现代 Web 应用的核心理念。

渐进增强能够为用户提供尽可能多的功能体验。这意味着开发的网站不会仅仅因为用户浏览器不支持某项功能就崩溃。

可以把渐进增强看作一种分层构建 Web 应用的方式。从基础的内容、简单的 HTML 链接、图像等开始。然后为用户提供 JavaScript 支持，添加一层增强的链接用于异步获取内容，并使用交互式的谷歌地图替换地图的静态图片；为支持 service worker 的浏览器添加离线支持；向可以接收推送的用户发送通知。

这样做的优点是，不仅可以为所有用户提供功能完整的应用，还可以使你的网站兼容所有用户（包括那些使用旧浏览器或者功能电话的用户），而且还可以让搜索引擎正确地索引所有内容。

在注册 service worker 时，我们首先要验证浏览器支持情况。支持 service worker 的浏览器用户将会享受到增强的体验，而其他用户仍将获得过往的全部体验。我们在逐步增强应用的同时，不会影响到其他的用户。

注意不要混淆**渐进增强**和**渐进式 Web 应用**这两个术语。虽然理想的做法是使用渐进增强的方式来开发渐进式 Web 应用，但是从技术上来说这不是必须的。你可以开发一个这样的渐进式 Web 应用，它在现代浏览器上运行得很好，但是在其他所有浏览器中都很糟糕——请不要这样做。

2.7　HTTPS和service worker

正如你刚才看到的，service worker 可以拦截请求、修改内容，甚至把内容完全替换成新的响应。为了保护用户和防止中间人攻击，避免恶意的第三方利用这些权限，只有使用安全连接（HTTPS）的页面才能注册 service worker。

在开发过程中，你可以通过主机名 localhost 使用 service worker，这样可以绕过安全连接的限制（例如，http://localhost/ 和 http://local-host:1234/user/index.html 都可以注册并使用 service worker）。但是一旦你把 Web 应用部署到服务端，就必须使用安全的 HTTPS 连接来保证 service worker 正常工作。

随着 Web 变得越发强大，它的许多新特性都要求使用 HTTPS。除了 service worker，其他很多新特性同样有这个要求。例如，SpeechRecognition 等其他 API 虽然没有强制要求使用 HTTPS，但是在 HTTPS 环境下，其功能要强大得多。甚至还有一些功能过去在非安全连接下可以使用，但是已经变成仅在 HTTPS 环境下可用了，例如 Geolocation API。

如果你还需要更多的动力以迁移到 HTTPS，Google 宣布它已开始在搜索排名中给予使用安全连接的网页略高的权重。

如今，网站使用 HTTPS 比以前更便宜，也更简单。很多新的证书机构甚至已经开始免费

提供 SSL 证书，而且配置服务器的新工具使得配置过程变得更加简单。如果你仍然坚持使用 HTTP，很快就会没有借口了。

2.8　从Web获取内容

在前面编写的代码中，我们通过指定内容和头部，从零开始创建了一个新的响应对象，并使用它来响应请求。

而 service worker 更广泛的用途是响应来源于网络的请求。

把 serviceworker.js 的代码替换成下列内容：

```
self.addEventListener("fetch", function(event) {
  if (event.request.url.includes("/img/logo.png")) {
    event.respondWith(
      fetch("/img/logo-flipped.png")
    );
  }
});
```

如果你遵循上述所有步骤，那么当你刷新页面时，网站的标志应该会上下翻转（见图 2-7）。

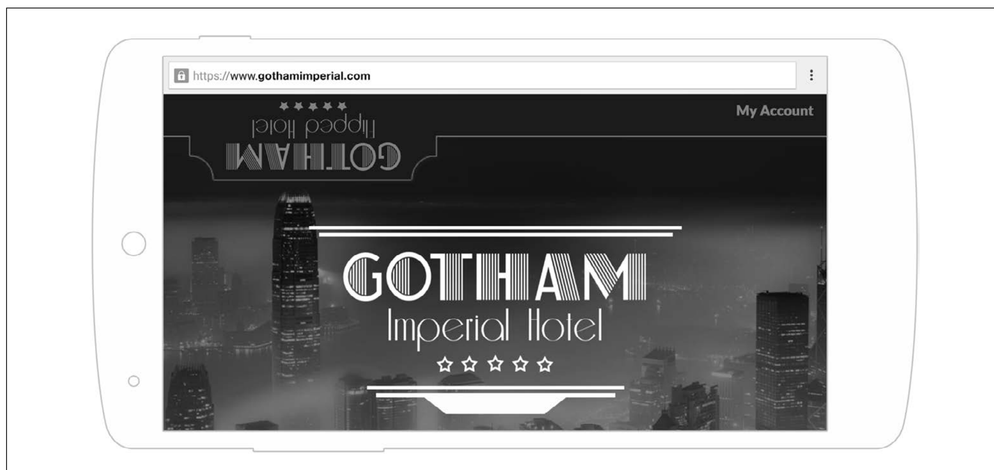


图 2-7：使用另一张图片来重写图片请求

和之前一样，我们监听 fetch 事件，只不过这次我们寻找的请求是 /img/logo.png。当检测到这样的请求时，我们使用 `fetch` 命令创建一个新的请求，并传递另一个标志图片的 URL。`fetch` 会返回一个 promise，其中包含了新的响应，我们在 `event.respondWith` 方法中使用它来响应原本的请求。

换句话说，我们让 service worker 监听了标志图片的请求，并请求另一个标志图片作为代替，然后返回给浏览器窗口。

fetch(request[, options]);

fetch 方法的第一个参数是强制必传的，可以包含一个 request 对象，也可以包含一个相对路径或者绝对路径的 URL 字符串：

```
// 通过URL请求
fetch("/img/logo.png");
// 通过request对象中的URL请求
fetch(event.request.url);
// 通过传递request对象请求
// 在这个request对象中，除了URL，可能还包含了额外的头部信息、表单数据等
fetch(event.request);
```

第二个参数是可选的，可以包含一个对象，里面是请求的选项。

下面这个例子发起了一个图片的 POST 请求，并在头部中包含了 cookie 信息（credentials 属性的默认值是 omit，这意味着 fetch 默认是不会发送 cookie 的）：

```
fetch("/img/logo.png", {
  method: "POST",
  credentials: "include"
});
```

fetch 会返回一个 promise，它在解析之后会得到一个响应对象。

2.9 捕获离线请求

让我们使用刚才学到的关于 service worker 的所有内容，来检测用户何时处于离线状态，并向他呈现友好的错误消息，用来代替浏览器的默认错误提示。

我们从修改 serviceworker.js 的代码开始，对于所有的请求，简单地获取并返回它们原始请求的内容：

```
self.addEventListener("fetch", function(event) {
  event.respondWith(
    fetch(event.request)
  );
});
```

仔细阅读前面的代码，你可能会想知道这个事件的意义是什么。我们监听并捕获了所有的 fetch 事件，然后使用另一个完全相同的 fetch 操作进行响应。如果你在浏览器中查看这个网站，会看到网站的行为和我们添加 service worker 之前是完全一致的。

那么这有什么意义呢？你可能还记得，在上一个例子中，我提到了 fetch 返回的响应是包裹在一个 promise 中的。通过 promise 包裹响应之后，我们就可以在 promise 失败的时候捕获异常，并进行处理。

把 serviceworker.js 的代码替换成下列代码：

```
self.addEventListener("fetch", function(event) {
  event.respondWith(
    fetch(event.request).catch(function() {
      return new Response(
      "Welcome to the Gotham Imperial Hotel.\n"+
      "There seems to be a problem with your connection.\n"+
      "We look forward to telling you about our hotel as soon as you go online."
      );
    })
  );
});
```

刷新浏览器，确保最新版本的 service worker 已经被正确注册与安装，然后切换到离线状态（参见 2.4 节的“模拟离线状态”）并再次刷新页面。现在，你应该会看到哥谭帝国酒店的个性化消息，而不是浏览器的本地错误提示（见图 2-8）。

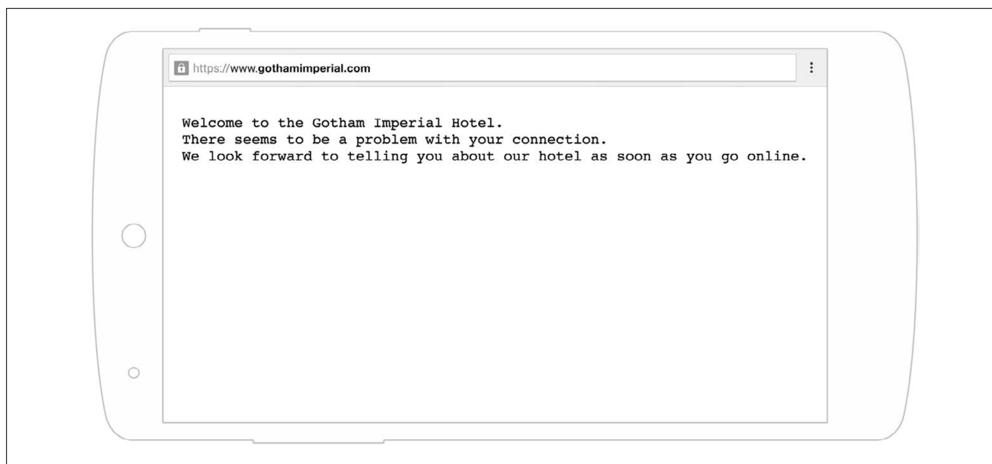


图 2-8：简单的离线文本信息

让我们给这个消息添加格式，并逐行检查代码。

2.10 创建HTML响应

由于我们正在推动 Web 向前迈进，而不是后退，所以我们需要改进代码，向离线用户发送一个优雅的 HTML 页面，而不是纯文本内容。

把 serviceworker.js 中的代码替换成下列代码：

```
var responseContent =
  "<html>" +
  "<body>" +
  "<style>" +
  "body {text-align: center; background-color: #333; color: #eee;}" +
  "</style>" +
  "<h1>Gotham Imperial Hotel</h1>" +
```

```
    "<p>There seems to be a problem with your connection.</p>" +
    "<p>Come visit us at 1 Imperial Plaza, Gotham City for free WiFi.</p>" +
    "</body>" +
    "</html>";

  self.addEventListener("fetch", function(event) {
    event.respondWith(
      fetch(event.request).catch(function() {
        return new Response(
          responseContent,
          {headers: {"Content-Type": "text/html"}}
        );
      })
    );
  });
```

刷新浏览器并确保注册了新的 service worker 之后，在离线状态下访问页面，检验应用新的离线消息（图 2-9）。

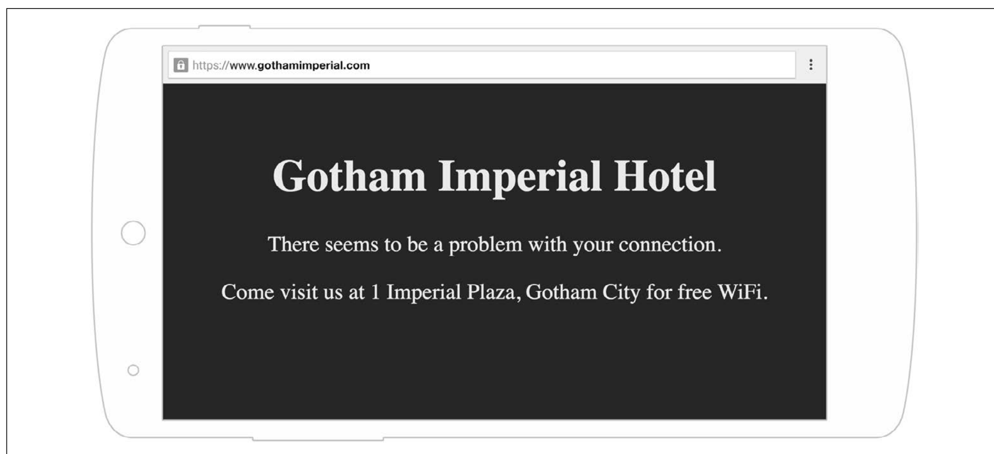


图 2-9：简单的离线 HTML 消息

让我们看看这是如何实现的。

首先，我们定义了希望显示给离线用户的 HTML 内容，并将其放进了一个名为 `responseContent` 的变量中。

然后添加了一个事件监听器，监听所有的 `fetch` 事件。当检测到 `fetch` 事件时，我们的函数会被调用，它接收一个 `FetchEvent` 对象作为参数。随后，我们调用该事件对象的 `respondWith` 方法来响应这个事件，避免其触发默认行为。

`respondWith` 方法接收一个参数，它可以是一个响应对象，也可以是一段可以通过 promise 得出响应对象的代码。后续的代码构建了该响应。

我们首先调用了 `fetch`，并传入原始请求（通过 `Fetch Event` 对象获取）。要强调的是，我们需要传入原始的请求对象，而不仅仅是 URL，以保证任何的头部信息、cookie 和请求方

法都保持不变。`fetch` 方法返回一个 promise。如果用户和服务器都在线，文件可以访问，并且整个过程一切顺利，那么这个 promsie 就会变成完成状态，`fetch` 会返回响应。然后响应会被传递回 `event.respondWith`，它再将响应发送回用户正在浏览的页面。但是，如果在试图获取文件的时候出错（例如用户登机了），promise 就会变成失败状态，我们在 `catch` 方法中定义的回调函数就会被调用。

在这个回调函数里，通过调用 `new Response` 构造了一个新的响应，并传入了两个参数。第一个参数是响应的主体（即我们之前定义的 HTML）。第二个参数是可选的选项对象。我们使用这个选项对象来添加一个 `Content-Type` 头部到响应中。

随后，这个新的响应被返回给 `event.respondWith`，并发送给页面，就像 Web 服务器返回了一个常规响应一样。



你可能会疑惑，为何我们一定要在创建响应的时候手动定义 `Content-Type` 头部。不妨尝试修改代码，把头部信息删掉，只返回响应内容，并观察结果。

浏览器将会把响应视为纯文本。所有的内容都会显示为纯文本，包括 HTML 标签和样式。

通常你不需要告诉浏览器 HTML 是 HTML，那么在这里发生了什么呢？

大部分 Web 服务器的配置，都会为大部分常见文件类型自动提供正确的头部信息。当服务器发送 HTML 文件时，会构造一个包含 HTML 和多个头部的响应，其中包括一个 `Content-Type` 头部，让浏览器知道该如何处理响应。由于我们是从头开始构造响应的，所以不仅要关心响应的内容（HTML），还要处理响应的头部。

2.11 理解service worker作用域

在本章前面，我们把 service worker 文件放在了项目的根目录中，现在我们来探究为什么一定要放在根目录而不是子目录中（例如 /js/sw.js）。

由于 service worker 功能强大，可以修改任何通过它的请求，因此需要对其进行一定的安全限制。

试想一下，你拥有一个网站，其中列出了哥谭最好的餐馆（例如 http://www.GothamEats.com/）。假设你允许每个餐馆在你的域名下托管一个网站，以提供餐馆的菜单和照片（例如 http://www.GothamEats.com/Ginnos）。如果 Ginnos 餐馆的所有者上传了一个 service worker 脚本到他的网站（例如 http://www.GothamEats.com/Ginnos/sw.js），它可以改变其竞争对手网站（例如 http://www.GothamEats.com/Ralphs）的所有流量，说该餐馆歇业了，会发生什么？假如浏览器有一天允许这样的情况发生，哥谭的秩序将会受到损害。

为了预防这种问题，每个 service worker 都有一个有限的控制范围。这个范围就是通过放置 service worker 的 JavaScript 文件的目录决定的。之前，我们通过把 serviceworker.js 文件放置在项目的根目录中，允许其控制来自站点中任何地方的所有请求。如果我们把它放置到 js 目录中，只有源于该子目录的请求才会通过它。

你可以在注册 service worker 的时候传入一个 scope 选项，用来覆盖 service worker 默认的作用域。这样做可以把 service worker 的作用域限制为目录的较小子集，但是不能扩展到比它的可用范围更广（例如，你可以限制位于 /ginnos/sw.js 的 service worker，让它只能影响到 /ginnos/menu/ 的请求，但是你不能将其作用域扩展到域的根目录）。

```
// 这两条命令将具有完全相同的作用域：
navigator.serviceWorker.register("/sw.js");
navigator.serviceWorker.register("/sw.js", {scope: "/"});

// 这两条命令将注册两个不同的service worker
// 每个service worker各自控制了一个不同的目录：
navigator.serviceWorker.register("/sw-ginnos.js", {scope: "/Ginnos"});
navigator.serviceWorker.register("/sw-ralphs.js", {scope: "/Ralphs"});
```

2.12 小结

虽然很容易认为我们仅仅是把浏览器的错误消息替换成了不那么奇特的错误消息[2]，但实际上，我们在这里已经完成了一个令人难以置信的壮举。

通过注册 service worker 并监听请求，我们将自己置于浏览器和网络之间。我们学会了如何拦截每一个页面请求（包括向第三方服务器发出的请求），以及如何修改、替换请求，或者检测请求失败的情况。

最重要的是，我们增强了网站的功能，使得离线用户不再处于黑暗之中。到达哥谭（这里的信号塔时不时会出问题）的用户，即使在离线状态下也能浏览一个简化的网站版本。

在下一章中，我们将利用所学到的 service worker 知识，再加上一点缓存“魔术”，为用户提供完整的哥谭帝国酒店体验，不管他们在线还是离线。

注 2：你知道可以在 Chrome 的错误页面上玩恐龙吗？快去试试点击它吧！

第 3 章

CacheStorage API

在第 2 章结尾，我们已经让 Web 和哥谭帝国酒店向前迈进了一大步。当用户离线时，我们向他们展示了自定义 HTML 内容，而不是浏览器的错误提示。不幸的是，我们同时后退了两步，因为我们只能展示一个简单的页面，没有图片，没有样式，也没有品牌介绍，使得这个现代化网站和“哥谭帝国酒店”这个名称不相称。

本章的目标是让用户在离线访问我们的站点时，可以看到 index-offine.html 文件的内容，包括其中的图片和样式（见图 3-1）。可以在浏览器中打开这个页面，看看它的样子（http://localhost:8443/index-offine.html）。



图 3-1：我们希望展示给离线用户的页面

你可能会猜测，为了实现这一点，我们必须捕获失败的请求，并返回代替的内容：

```
self.addEventListener("fetch", function(event) {
  event.respondWith(
    fetch(event.request).catch(function() {
      return fetch("/index-offline.html");
    })
  );
});
```

你能发现这段代码的问题吗？

代码虽然没有编码错误，但是有逻辑问题。我们只有在知道用户离线的时候，才试图去获取离线文件。我们需要在用户在线的时候先拿到文件，并存储在设备上，这样在用户离线的时候才能提供内容。

现在仍然未解的一个谜团是在用户设备上用来存储内容的神秘领域。所幸的是，当 service worker 被引入的时候，我们还得到了新的 CacheStorage API——这正是那个未解的谜团。

3.1　CacheStorage是什么，不是什么

CacheStorage 是一种全新的缓存层，你拥有完全的控制权。

我们都知道老式的浏览器缓存。这种缓存在后台不倦地工作，决定哪些文件需要缓存，何时从缓存或者网络获取文件，以及何时删除旧的缓存文件。作为开发人员，这完全不在你的控制范围内。你影响浏览器缓存内容的唯一方式，就是通过 HTTP 头部（在服务器发送每个响应时一并发送）向浏览器提示相关的内容。

CacheStorage 也不像旧的 AppCache API，后者使用了一种更落后、更死板的方式，通过缓存清单文件定义了哪些文件应该能够脱机使用。AppCache API 已经从 Web 标准中移除，并且 Jake Archibald 在文章“Application Cache is a Douchebag”中猛烈地抨击了它。

CacheStorage 采取了不同的方式，将控制权放到了开发人员的手中。

和前面提到的技术不同，这是通过暴露一系列基本的方法（如创建和打开任意数量的缓存，以及在其中存储、检索或删除响应）来实现的。

将 service worker 和 CacheStorage 的能力结合在一起，我们可以通过程序直接控制缓存哪些内容、删除哪些缓存，以及哪些内容从缓存返回、哪些内容从网络返回。

3.2　决定何时进行缓存

让我们回到哥谭帝国酒店，看看应该如何缓存文件，以显示我们的离线网站。

我们已经知道，在用户离线时尝试获取索引文件的离线版本会引起问题。我们真正需要的，是在知道用户在线时，就去获取这个文件以及其他相关的文件。

让我们来看看简化版的 service worker 生命周期（见图 3-2）。

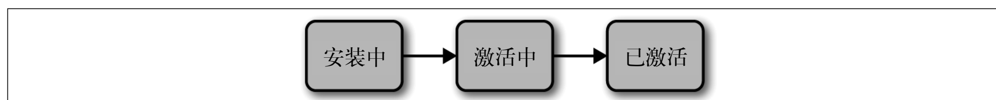


图 3-2：service worker 生命周期的简化表示

到目前为止，我们只使用 service worker 监听了 fetch 事件，这类事件只能够被**激活**状态的 service worker 所捕获。我们需要监听一个较早发生的事件，并使用这个事件来缓存我们的 service worker 所依赖的文件。

为此，我们可以使用 service worker 的 install 事件。在每个 service worker 的生命周期中，这个事件只会发生一次，即在首次注册之后以及激活之前发生。在 service worker 接管页面并开始监听 fetch 事件之前，我们通过监听这个事件，得到了一个极好的机会来缓存所有希望离线可用的文件。

如果出现问题，我们甚至可以在 install 事件中取消安装 service worker。这使得安装阶段成为了缓存所需请求的绝佳机会。如果在缓存时出现问题，可以中止安装，因为我们知道，浏览器会在用户下次访问页面时再次尝试安装 service worker。通过这种方式，我们可以有效地为 service worker 创建安装依赖——在 service worker 安装并激活之前，必须先下载并缓存这些文件。

3.3 在CacheStorage中存储请求

让我们开始编码。如果你没有完成第 2 章中的所有步骤，或者想要确保当前代码处于本章开头的状态，请在命令行中运行以下内容：

```
git reset --hard
git checkout ch03-start
```

清空 serviceworker.js 文件的内容，并替换为以下代码：

```
self.addEventListener("install", function(event) {
  event.waitUntil(
    caches.open("gih-cache").then(function(cache) {
      return cache.add("/index-offline.html");
    })
  );
});
```

此处有一些新的命令是我们之前没有遇到过的。让我们逐一检查。

首先，我们为 `install` 事件添加了事件监听器。在新的 service worker 注册之后，这个事件会立即在其安装阶段被调用。

由于我们的 service worker 将依赖于 index-offline.html，我们需要先验证它是否已经成功缓存，然后才能认为安装成功，并激活新的 service worker。因为需要异步获取文件并缓存起来，所以我们需要延迟 `install` 事件，直到异步事件完成。

为了实现这一点，我们在 `install` 事件中调用了 `waitUntil`。`waitUntil` 会延长事件的存在时间，直到传入的 promise 得以解决。这样我们就可以等到成功将文件存储在缓存中，再声明 `install` 事件完成，并且，如果在任何步骤遇到了问题，可以通过拒绝 promise 中止安装。

在 `waitUntil` 函数中，我们调用了 `caches.open` 并传入了缓存的名称（这提示了 CacheStorage 的另一个强大特性：我们可以为网站创建多份缓存，在第 4 章我们将会利用这一特性）。

caches.open 打开并返回一个现有的缓存，或者如果没有找到对应名称的缓存，就将创建并返回它。caches.open 返回一个包裹在 promise 中的 cache 对象，因此我们接下来使用 then 语句，并传入一个函数，该函数将这个 cache 对象作为参数。

我们需要做的最后一件事情，是调用 cache.add('/index-offline.html')，这个方法将请求文件并将文件放入缓存中，缓存的键名就是 "/index-offline.html"。

前面例子中的代码把一系列的 promise 串联在一起。粗略翻译成伪代码，可以这样表达：

```
If 检测到install事件，需要先完成下列内容才能宣布成功：
  首先你要成功打开缓存
    随后
  你要请求文件，并存储在缓存中
如果上述步骤中的任何一步失败，则中止service worker的安装。
```

在 install 事件中，通过 waitUntil 等待缓存完成，确保了在整个链条中如果遇到任何问题，service worker 都不会被安装。在已经激活的 service worker 的任何代码中，我们都可以认为安装事件已经成功完成了，并且 index-offine.html 是在缓存中可用的。

3.4 从CacheStorage中取回请求

既然已经将页面的离线版本存储到 CacheStorage 当中，我们需要从缓存中取回并返回给用户。

在 serviceworker.js 中添加下列代码，放置到监听 install 事件的代码的后面：

```
self.addEventListener("fetch", function(event) {
  event.respondWith(
    fetch(event.request).catch(function() {
      return caches.match("/index-offline.html");
    })
  );
});
```

你可能会觉得这段代码似曾相识，和上一章的代码很相似。唯一的区别是，我们没有构造一个新的响应或者从 Web 获取内容，而是通过调用 caches.match，从 CacheStorage 中返回内容。

你可能还注意到，代码从缓存中返回请求的时候，甚至没有先验证它是否存在于缓存中。这是因为我们已经让 service worker 的安装强依赖于缓存这一请求。



CacheStorage 遵循了同源安全策略。无论你是使用 caches.match() 还是 caches.open()，都只能访问当前源创建的缓存。换句话说，当你的应用运行 caches.match("bank-password") 的时候，只有你的应用创建的缓存可以被搜寻到。要了解更多关于同源策略的内容，参见 10.2.2 节中的“同源策略”。

match(request[, options]);

给定一个请求，match 方法会从缓存中返回一个 response 对象。

match 方法可以在 caches 对象上调用，这样会在所有缓存中寻找，也可以在某个特定的 cache 对象上调用：

```
// 在所有缓存中寻找匹配的请求
caches.match("logo.png");

// 在特定的缓存中寻找匹配的请求
caches.open("my-cache").then(function(cache) {
  return cache.match("logo.png");
});
```

match 方法的第一个参数是需要在缓存中寻找的内容，可以是 request 对象或者 URL。这应该和你添加到缓存中的请求相匹配。

第二个参数是非必传的选项对象。

match 会返回一个 promise，并向 resolve 方法传入在缓存中找到的第一个 response 对象，当找不到任何内容的时候，它的值是 undefined。

即使在找不到对应的响应时，match 方法返回的 promise 也不会被拒绝。出于这个原因，除非你可以确保肯定存在匹配，否则可能需要在返回之前先判断是否找到了匹配：

```
caches.match("/logo.png").then(function (response) {
  if (response) {
    return response;
  }
});
```

3.5 在示例应用缓存

如果你再次访问首页（让这个新的 service worker 有机会安装），然后在离线状态下又一次访问，应该能够看到 index-offline.html 的内容。

但我们还没有完成。我们的代码现在只知道如何缓存并提供单个文件——index-offline.html。由于没有样式和图片，我们为离线用户提供的体验非常糟糕。此外，我们的代码在任何请求失败的情况下，都会傻傻地缓存并返回同一个 HTML 文件。不管用户请求的是 index.html、bootstrap.min.css 还是 gih-offine.css，只要任何请求失败，service worker 总是会返回相同的 HTML 文件（见图 3-3）。



图 3-3：请求样式表时返回的 HTML

在宣告大功告成之前，我们还需要改进 service worker，把它需要的所有资源都存储到缓存中，并将每个请求与正确的响应相匹配。

让我们从缓存 index-offine.html 的所有样式和图片文件开始。

我们可以利用迄今为止所学到的一切来完成这项任务，简单地串联一系列 `cache.add` 调用即可：

```
self.addEventListener("install", function(event) {
  event.waitUntil(
    caches.open("gih-cache").then(function(cache) {
      return cache.add("/index-offline.html").then(function() {
        return cache.add(
          "https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css"
        );
      }).then(function() {
        return cache.add("/css/gih-offline.css");
      }).then(function() {
        return cache.add("/img/jumbo-background-sm.jpg");
      }).then(function() {
        return cache.add("/img/logo-header.png");
      });
    })
  );
});
```

这样做……并不优雅。

上述代码不仅不优雅，而且通过这样的链式调用，每一个文件在请求并缓存之前，必须等待上一个请求完成。这样做使得 service worker 的安装过程缓慢。

幸运的是，还有更佳的方法。

在 serviceworker.js 中，将 `install` 事件监听方法替换成下列代码：

```
var CACHE_NAME = "gih-cache";
var CACHED_URLS = [
  "/index-offline.html",
  "https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css",
  "/css/gih-offline.css",
  "/img/jumbo-background-sm.jpg",
  "/img/logo-header.png"
];

self.addEventListener("install", function(event) {
  event.waitUntil(
    caches.open(CACHE_NAME).then(function(cache) {
      return cache.addAll(CACHED_URLS);
    })
  );
});
```

在上述示例中，我们首先设置了两个新的变量。第一个变量包含了缓存的名称，第二个变量是一个数组，其中包含了一份需要存储的 URL 列表。

接下来，我们使用 `cache.addAll()` 代替了 `cache.add()`，并传入需要缓存的 URL 数组。

`cache.addAll()` 的作用和 `cache.add()` 类似，但是前者接收的不是单个 URL，而是一个 URL 数组，并全部存储到缓存中。如果任何一个请求失败，类似于 `cache.add()`，`cache.addAll()` 返回的 promise 将会被拒绝。

3.6　匹配每个请求的正确响应

现在我们缓存了展示离线应用所需要的所有静态资源，但是对于每个失败的请求，我们仍然盲目地返回 index-offine.html 的内容。甚至在请求一张图片的时候也会返回 HTML。

我们需要将每个失败的请求与正确的缓存响应相匹配，并提供给用户。

在 serviceworker.js 中，把 `fetch` 事件监听方法替换成下列代码：

```
self.addEventListener("fetch", function(event) {
  event.respondWith(
    fetch(event.request).catch(function() {
      return caches.match(event.request).then(function(response) {
        if (response) {
          return response;
        } else if (event.request.headers.get("accept").includes("text/html")) {
          return caches.match("/index-offline.html");
        }
      });
    })
  );
});
```

在这个示例中，新的 `fetch` 事件处理代码依然试图向网络发起请求，并将响应返回给在线用户。但是，如果有任何请求失败，`catch` 块中的新函数就会起作用。

catch 函数一开始会试图将发起的请求与在任何缓存中存储的请求相匹配。由于 caches.match 返回的 promise 是不会被拒绝的（即使匹配不成功也是如此），在返回之前，我们先要使用 if (response) 来检查是否在缓存中找到了响应。如果没有找到，我们将直接返回 index-offine.html 的内容作为代替。请记住，浏览器永远不会显式地请求 index-offine.html，它可能会请求 /index.html 或者是根目录（/）。我们只需要返回 index-offine.html 作为代替即可。

为了安全起见，我在返回 index-offine.html 之前，还进行了一项额外的检查。这项检查确保了请求是包含了 text/html 的 accept 头部的。这样可以确保我们不会返回 HTML 内容给其他类型的请求，例如对图片、样式表的请求等。[1]

你可能还记得，当我们从头开始创建一个新的 HTML 响应时，必须将其 Content-Type 的值定义为 text/html，以便浏览器可以正确地将响应识别为 HTML。那为什么我们在上述代码示例中可以直接返回响应，无须手动定义 Content-Type 为 HTML、CSS 或者图片呢？其原因是，cache.add() 和 cache.addAll() 请求并缓存的是一个完整的 response 对象。这个对象不仅包含了响应体，还包含了服务器返回的任何响应头（其中包含了 Content-Type）。



ignoreSearch

通过传递 request 对象（例如 caches.match(event.request)）来查找缓存中的条目有一个潜在陷阱，你应该牢记于心。

用户可能不会总是使用相同的 URL 来访问你的网站。例如，假设你正在网站上运行一个新的推广活动。如果用户从网站主页点击来到这个活动的页面，那么他访问的链接将是 https://www.site.com/promo.html。但是如果用户通过点击横幅广告来到这个活动的页面，他访问的链接可能是 https://www.site.com/promo.html?utm_source=halloween-campaign&utm_medium=cpc。存在数百种 URL 变体的情况并不罕见，每条 URL 的差异仅仅在于它们的查询字符串（也称为 search 属性，还可以理解为 URL 问号后面的一串难以阅读的字符）。

如果你在 install 事件中把 /promo.html 保存在缓存中，然后使用 caches.match(event.request) 尝试去寻找 /promo.html?utm_source=a，会找不到任何内容。为了解决这个问题，你可以编写特定的规则，在将查询字符串传递给 match() 之前，将其从 URL 中剥离。或者检测 URL 字符串中是否包含了 promo.html，然后传递一个硬编码的 URL 给 match()。但是，还有更好的办法。

如果你可以确保查询字符串对于页面内容不会产生影响，可以使用 ignoreSearch 选项，通知 match() 方法忽略查询字符串：

```
caches.match(event.request, {ignoreSearch: true})
```

这样将会匹配到请求 URL 的条目，同时会忽略查询参数（例如 /promo.html 可以同时匹配 /promo.html?utm_source=urchin 和 /promo.html?utm_medium=social）。

注 1：感谢 Jeffrey Posnick 提出的建议，具体参见 https://pwabook.com/matchhtml。

3.7 HTTP缓存和HTTP头

需要记住的重要一点是，CacheStorage 不能取代过去的 HTTP 缓存。

如果你的服务器提供的文件包含了一个 HTTP 头，说明该文件可以在浏览器缓存中保存一年时间，浏览器就会一直使用浏览器缓存来提供这个文件。当你试图在 service worker 中请求一个文件的时候，在发起网络请求之前，它依然会先检查浏览器缓存。

我们来看一个例子。

在 service worker 安装的时候，它调用了 `cache.addAll(['/main.css'])`。随后，这个文件会从网络请求回来，并保存在 CacheStorage 中。如果服务器提供这个文件时包含了头部信息 `Cache-Control: max-age=31536000`（服务器通过这种方式表示文件可以缓存一年时间），除了 service worker 会将文件保存在 CacheStorage 之外，文件也会被保存在浏览器缓存中。

如果你在一周之后更新了 main.css，并打算更新 service worker，让其重新调用 `cache.addAll(['/main.css'])`，那么该文件会从浏览器缓存而不是网络中返回。

这并非 CacheStorage 特有的问题。HTTP 缓存一直都是这样工作的。如今，理解并正确实现 HTTP 缓存和过去一样重要。关于该主题的入门介绍，可以参见 Jake Archibald 的文章“Caching best practices & max-age gotchas”。

3.8 小结

本章完成了很多工作，我们获得了一个强大的新工具 CacheStorage，并学习了如何创建一个可以针对不同请求提供不同内容的 service worker。我们理解了如何请求并缓存响应，以及如何为 service worker 的安装过程创建依赖。最后，我们结合了所有这些新工具，为用户提供了一个现代化的、带有品牌烙印的离线版本首页（见图 3-4）。



图 3-4：哥谭帝国酒店的离线品牌页面

我们可以更进一步，将新技能与第 2 章内容结合起来，在用户在线和离线的情况下，都返回缓存内容。对于不经常改变的内容，这种方式是非常有用的，可以显著加快网站的加载速度，并能节省用户的带宽与我们的服务器成本——但是在这里我就不作展开了。

下一章，我们将从所有这些兴奋点中脱离出来，花一些时间正确地理解 service worker 的生命周期。

我们在熟悉了 install 事件之后，就可以创建安装依赖。与此类似，在理解 service worker 生命周期的其他部分之后，我们就可以强而有力地掌控我们的渐进式 Web 应用了。

我们还会花时间研究开发者工具，因为它们可以使开发人员的工作变得轻松。在下一章的最后，作为负责任的成年人，我们将承担责任，学习管理缓存。这将会相当有趣。

第 4 章

service worker生命周期和缓存管理

现在你已经可以试着使用 service worker 了，可能你会注意到，它的行为有一些特殊性。

有时候，在加载页面时，你的 service worker 似乎在控制页面；有时候，你却需要先刷新页面（即使 service worker 处于激活状态）；甚至在一些场景下，你更改了 service worker 的代码之后，却发现不管刷新页面多少次，都没有发生变化。

在第 2 章中，我鼓励你打开了"Update on reload"选项，它让你在每次页面刷新后能够立即看到 service worker 的任何更改。然而，这就像老式的视频游戏作弊代码一样，方便的处理方式虽然让事情变得简单，但并不能代表现实世界中的事物是如何运作的。

service worker 的特殊性起初可能会让人疑惑，然而，一旦你了解了 service worker 的状态变化流程，一切谜团就将解开。



本章探索并使用了浏览器中的许多开发者工具。简便起见，本章将假设你正在使用 Chrome 访问应用。我们的代码在支持 service worker 的所有浏览器中都能运行，但是开发者工具的位置和可用性可能因为浏览器类型和浏览器版本的不同而有所差异。

参见 4.8 节获取更多细节。

让我们来看看用户是如何体验我们的应用的。

在开始之前，请先确保你的代码处于第 3 章结束时的状态。在命令行中运行下列命令：

```
git reset --hard
git checkout ch04-start
```

如果项目中的本地服务器还没有运行，可以通过 `npm start` 命令启动。

把 serviceworker.js 中的代码替换成下列代码：

```
self.addEventListener("install", function() {
  console.log("install");
});

self.addEventListener("activate", function() {
  console.log("activate");
});

self.addEventListener("fetch", function(event) {
  if (event.request.url.includes("bootstrap.min.css")) {
    console.log("Fetch request for:", event.request.url);
    event.respondWith(
      new Response(
        ".hotel-slogan {background: green!important;} nav {display:none}",
        { headers: { "Content-Type": "text/css" }}
      )
    );
  }
});
```

这段代码现在对你来说应该相当熟悉了。它监听了 `install` 和 `activate` 事件（本章后面将对它进行探究），并在这两个事件被触发的时候将消息记录到控制台。它还监听了 `fetch` 事件，在请求 `bootstrap.min.css` 的时候，把响应内容变成简单的样式表，其中把页眉的背景色改成了绿色。

你自然会认为，在访问我们的应用时，将会看到绿色的背景。在验证这个假设之前，你需要确保像第一次访问的用户那样体验这个应用。

(1) 在浏览器中打开应用（http://localhost:8443/）。

(2) 如果“Update on reload”打开了，将其关闭（参见 2.5 节中的“service worker 生命周期”）。

(3) 删除所有已经注册到页面中的 service worker。在 Chrome 中，这可以使用在开发者工具的 Application 面板中的“Clear storage”工具来完成。参见 4.8 节以获取更多详情。

　　通过删除 service worker，你就可以确保在下次访问时，作为一个还没有安装 service worker 的新用户来访问页面。

刷新页面。页面看起来应该如图 4-1 所示。

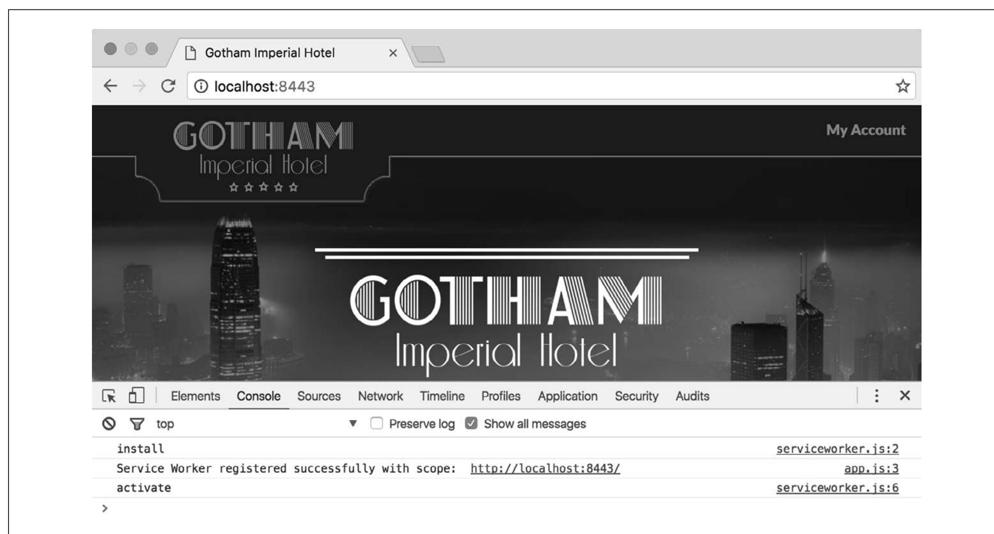


图 4-1：service worker 被激活，但是依然没有控制页面

再一次刷新页面。页面看起来应该如图 4-2 所示。

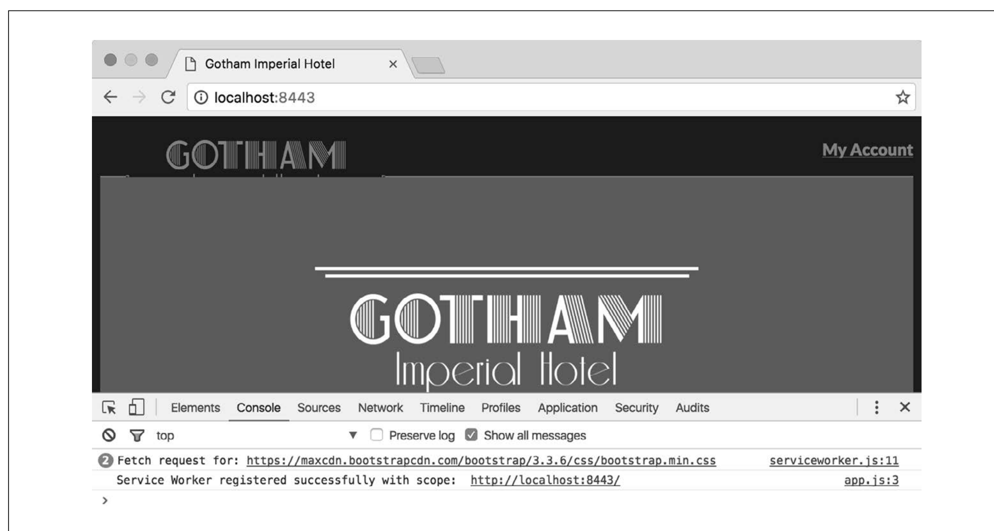


图 4-2：service worker 被激活，并控制了页面

这里发生了什么？正如你在图 4-1 中可以清楚看到的那样，service worker 在第一次刷新之后，成功安装并激活，但是没有捕获到 `fetch` 事件，导致样式表没有发生变化。为什么 service worker 需要二次刷新才能开始监听 `fetch` 事件？

要搞清楚这些问题，就需要理解 service worker 的生命周期。

4.1 service worker生命周期

当页面注册一个新的 service worker 的时候，service worker 会经历多个状态（见图 4-3）。

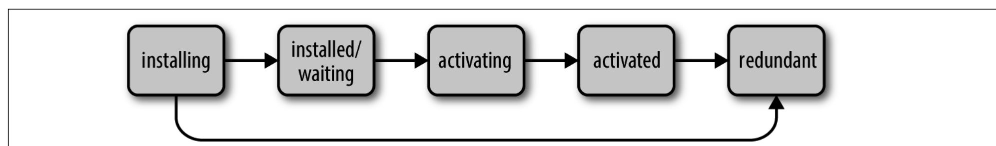


图 4-3：service worker 生命周期

`installing`（正在安装）

当你使用 navigator.serviceWorker.register 注册一个新的 service worker 时，JavaScript 代码就会被下载、解析，并进入安装状态。如果安装成功，service worker 就会进入到 `installed`（已安装）状态。但是，如果在安装过程中发生了错误，脚本将被永久地放逐到 `redundant`（废弃）状态的深渊中（你也可以在刷新页面之后重新注册它）。

安装过程的生命周期是可以延展的，只需要通过监听 `install` 事件，然后在其中调用 `waitUntil()` 方法，并传入一个 promise 即可。在这个 promise 完成或者失败之前，service worker 是不会认为安装完成的。如果 promise 失败了，整个安装过程就会失败，从而导致 service worker 变成 `redundant`（废弃）状态。



在第 3 章中我们创建了一个依赖关系，service worker 的成功安装依赖于静态资源的成功缓存。通过在 `install` 事件中调用 `waitUntil`，使得 service worker 在变成 `installed` 状态之前，需要先等待我们的缓存方法返回的 promise 完成，这样就可以确保，如果任何文件没有被缓存，安装过程就会失败，service worker 马上会进入 `redundant`（废弃）状态。

`installed/waiting`（已安装/等待中）

一旦 service worker 安装成功，就会进入 `installed` 状态。一般情况下，它会马上进入 `activating`（激活中）状态，除非另一个正在激活的 service worker 依然在控制应用，在这种情况下它会维持在 `waiting`（等待中）状态。

我们将会在 4.3 节中探究 `waiting` 状态。

`activating`（激活中）

在 service worker 激活并接管应用之前，会触发 `activate` 事件。和正在安装状态类似，`activating` 状态也可以通过调用 `event.waitUntil()` 并传入 promise 来进行扩展。

4.4 节将介绍如何利用这个事件来管理应用的缓存。

`activated`（已激活）

一旦 service worker 激活，它就准备好接管页面并监听功能性事件了（例如 `fetch` 事件）。

service worker 只能够在页面开始加载之前控制页面。这意味着，如果页面在 service worker 激活之前开始加载，service worker 是不能够控制页面的。我们会在后文中的

"为什么 service worker 不能在页面开始加载之后控制页面？"探索其中的原因。

`redundant`（废弃）

如果 service worker 在注册或者安装过程中失败，或者被新的版本替换，会被置为 `redundant` 状态。处于这种状态下的 service worker 将不再对应用产生任何影响。



请记住，service worker 及其状态是独立于任何一个浏览器窗口或者标签页的。这意味着一旦 service worker 处于 `activated` 状态，它就将保持在这个状态。即使用户打开第二个标签页，尝试再次注册这个 service worker 也是如此。如果浏览器检测到你试图注册的 service worker 已经激活，就不会再一次安装了。

可以确信，在一个 service worker 的生命周期中，安装和激活事件都只会运行一次。

进一步熟悉了 service worker 经历的各种状态后，让我们尝试理解为什么在第一份示例代码中，service workerr 在第二次刷新之前都没有改变应用的样式。

当用户第一次访问我们的站点（我们通过删除 service worker 后刷新页面的方式进行了模拟）时，应用会注册 service worker。service worker 的文件将会被下载，然后开始安装。`install` 事件被调度，然后触发了我们的函数，将调用的时机记录到控制台中。service worker 随后进入 `installed` 状态，然后立即变成 `activating` 状态。此时我们的另一个函数被触发，这次轮到了 `activate` 事件把状态记录到控制台中。最后，service worker 进入 `activated` 状态。现在它已经激活了，并准备好在其控制范围内控制页面。

不幸的是，当 service worker 正在安装的时候，我们的页面已经开始加载并渲染了。这意味着即使 service worker 变成了 `active` 状态，也不能够控制页面了。只有当我们刷新页面之后，我们的激活态 service worker 才能控制它。此时，service worker 既是激活的也能控制页面，并且可以监听和操控 `fetch` 事件。



为什么 service worker 不能在页面开始加载之后控制页面？

让我们来考虑另一种可能。假设一个 service worker 负责检测视频文件是否加载太慢，并提供其他镜像服务器的相同视频链接作为代替。这个 service worker 会拦截所有视频文件的请求，然后返回所请求的视频或者包含了镜像站点链接的 JSON 文件。这个 service worker 还会拦截 app.js 的请求，并提供代替版本的 app-sw.js。其中后者可以显示视频响应，并从 JSON 响应中渲染一份链接列表。现在设想一下，如果允许 service worker 控制那些加载后才注册 service worker 的页面会发生什么。如果在 service worker 获得控制权之前，页面下载了未修改的 app.js 文件，然后在请求视频的时候开始接收 service worker 返回的 JSON 文件，会发生什么呢？ app.js 并不知道如何处理这些响应，然后整个页面可能会崩溃。

通过确保每个页面仅由一个 service worker 全程控制（从开始加载到关闭），service worker 就能帮我们避免这些意外的问题。

4.2 service worker的生命周期与waitUntil的重要性

一旦 service worker 成功安装并激活，会发生什么呢？既然 service worker 不是直接绑定在任何标签页或者窗口上，并且可以随时响应事件，这是否意味着它一直在运行呢？

答案是否定的。浏览器没有让当前已注册的所有 service worker 一直保持运行的状态。如果这样做，随着越来越多的网站注册越来越多的 service worker，性能将很快受到影响，所有的 service worker 都必须保持一直运行。

代替方案是，service worker 的生命周期直接与它所处理的事件的执行联系在一起。当某个 service worker 作用域下的事件被触发，service worker 将被唤醒，处理事件，然后终止。

换句话说，当用户访问网站时，浏览器就会开始控制 service worker，一旦处理完来自页面的事件，它就终止了。如果稍后发生了另一个事件，service worker 将会再次启动，并在完成后立即终止。

如果我们在 service worker 的事件处理代码中异步调用了一些代码，会发生什么呢？举个例子，让我们看看下面这个 `push` 事件处理方法。第 10 章会详细讨论 `push` 事件，现在你只需要知道，在服务器推送消息给用户时，它将会被触发（甚至可能会发生在应用没有运行的时候）：

```
self.addEventListener("push", function() {
  fetch("/updates")
  .then(function(response) {
    return self.registration.showNotification(response.text());
  });
});
```

当 `push` 事件触发的时候，上述示例代码中的事件监听器将会尝试从服务端 `fetch` 更新，然后一旦接收到响应，就会向用户显示这些更新的通知。

但是这段代码有个问题。当 `fetch` 请求去异步查询更新的时候，事件监听器已经停止执行了。一旦事件结束，在响应返回之前，service worker 就会被浏览器终止。这就会导致没法处理响应和显示通知了。

我们怎样才能在让浏览器终止 service worker 之前，让 service worker 等待某件事情的发生呢？答案不言而喻。正如我们已经确定的那样，service worker 的生命周期是和它所处理的事件执行直接相关的，所以我们所需要做的就是通过 `waitUntil` 方法扩展其事件的执行：

```
self.addEventListener("push", function() {
  event.waitUntil(
    fetch("/updates")
    .then(function() {
      return self.registration.showNotification("New updates");
    })
  );
});
```

这段示例代码告知 `push` 事件等待我们传入的一个 promise 完成或者失败，才能认为该事件已完成。这意味着 service worker 的生命周期也得到了延长。最终的结果是，service worker 会一直停留，直到 `fetch` 和 `showNotification` 调用都完成为止。

4.3 更新service worker

下面看看尝试更新现有的 service worker 会发生什么。

修改 serviceworker.js 文件，将设置的头部背景色从原本的绿色（`green`）改为红色（`red`）。

你的 `fetch` 事件监听器现在应该看起来像这样：

```
self.addEventLi stener("fetch", function(event) {
  if (event.request.url.includes("bootstrap.min.css")) {
    console.log("Fetch request for:", event.request.url);
    event.respondWith(
      new Response(
        ".hotel-slogan {background: red!important;} nav {display:none}",
        { headers: { "Content-Type": "text/css" }}
      )
    );
  }
});
```

然后刷新页面，一次，两次，三次。

你可能会感到惊讶，你对 service worker 所做的修改没有影响到页面，背景依然是绿色的。

发生了什么呢？既然背景是绿色的，那么页面显然是由 service worker 所控制的，然而 service worker 文件却清楚地描述了它是红色的。

我们可以通过查看 Chrome 开发者工具中的 Application → Service Workers 部分，来理解这段示例代码中发生的事情（见图 4-4）。

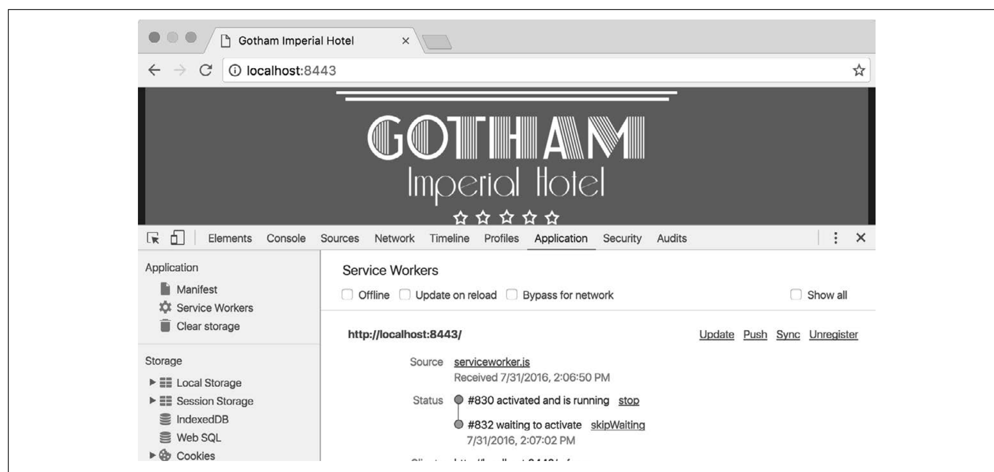


图 4-4：新的 service worker 正等待当前激活的 service worker 释放控制权

正如你在图中看到的那样，页面上注册了两个 service worker，但是只有其中的一个在控制页面。旧的 service worker（绿色背景色）是激活的，而新的 service worker（红色背景色）仍然处于等待状态。

每当页面加载一个激活的 service worker，就会检查 service worker 脚本的更新。如果文件在当前的 service worker 注册之后发生了修改，新的文件就会被注册和安装。安装完成后，它并不会替换现有的 service worker，而是会保持在 `waiting` 状态。它将一直停留在这个状态，直到 service worker 作用域中的每个标签页和窗口关闭，或者导航到一个不在其控制范围内的页面。只有在当前激活的 service worker 控制的页面全部关闭之后，这个旧的激活的 service worker 才会进入废弃状态，然后新的 service worker 才会激活。

这就解释了为何我们的应用背景没有发生改变。尝试关闭标签页，然后重新打开，或者导航到一个不同的网站，然后点击返回按钮。这样应该就能够让旧的 service worker 变成废弃状态，并且激活新的 service worker，然后背景色最终将会变成红色。



为什么安装完成的新 service worker，在接管控制并成为激活的 service worker 之前，必须等待作用域中的所有页面关闭呢？

假设有两个打开的标签页是由同一个 service worker 控制的。现在，如果刷新第一个标签页，下载一个新的 service worker 并激活它，会发生什么呢？第二个页面本来加载了旧的 service worker，突然被另外一个 service worker 所控制了。这样可能会导致很多意想不到的问题，比如我们在“为什么 service worker 不能在页面开始加载之后控制页面？”中探索的那个问题。

但是，在新的 service worker 安装完成后，为什么不能让新的 service worker 控制新的页面，同时让旧的 service worker 控制旧的页面呢？为什么浏览器不能跟踪多个 service worker 呢？为什么所有的页面都必须由单一的 service worker 所控制呢？

我们来探索这种场景引发的一种潜在灾难。设想这样一种场景：你发布了一个新版本的 service worker，这个 service worker 的 `install` 事件会从缓存中删除 user-data.json 文件，并添加 users.json 作为代替，并且修改 `fetch` 事件，让其在请求用户数据的时候，返回新的文件。如果多个 service worker 分别控制了不同的页面，旧 service worker 控制的页面可能会在缓存中搜索旧的 user-data.json 文件，但是这个文件已经被删除了，会导致应用崩溃。

通过确保所有打开的标签页从开始加载到关闭都由同一个 service worker 所控制，就可以避免这样的问题。这使得在任何时间都可以知道哪个 service worker 在控制着页面。

4.4 为什么需要管理缓存

现在我们已经了解 service worker 的生命周期，让我们回到应用中，看看当需要更新应用时会发生什么。

假设我们决定修改离线首页的内容。如果我们更新 sw-index.html 文件的内容，如何让 service worker 知道我们需要下载新版本的文件，并存储在 CacheStorage 中呢？

在开始之前，我们先在命令行中运行下列命令，把 serviceworker.js 恢复到第 3 章结束时的状态：

```
git reset --hard
git checkout ch04-start
```

serviceworker.js 文件的内容应该看起来像这样：

```
var CACHE_NAME = "gih-cache";
var CACHED_URLS = [
  "/index-offline.html",
  "https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css",
  "/css/gih-offline.css",
  "/img/jumbo-background-sm.jpg",
  "/img/logo-header.png"
];

self.addEventListener("install", function(event) {
  event.waitUntil(
    caches.open(CACHE_NAME).then(function(cache) {
      return cache.addAll(CACHED_URLS);
    })
  );
});

self.addEventListener("fetch", function(event) {
  event.respondWith(
    fetch(event.request).catch(function() {
      return caches.match(event.request).then(function(response) {
        if (response) {
          return response;
        } else if (event.request.headers.get("accept").includes("text/html")) {
          return caches.match("/index-offline.html");
        }
      });
    })
  );
});
```

请记住，我们的 service worker 会在安装阶段下载并缓存所需要的文件。如果希望它再次下载并缓存某些文件，就需要触发另一个安装事件。正如我们在 4.1 节中看到的那样，service worker 文件的任何修改都会导致下次访问应用的任何页面时，安装新的 service worker。



现在你已经了解 service worker 的生命周期，可以随时打开“Update on reload”开关以方便开发。

在 serviceworker.js 文件中，把第一行的缓存名称改成 `gih-cache-v2`。现在第一行应该看起来如下所示：

```
var CACHE_NAME = "gih-cache-v2";
```

通过给缓存名称添加版本号，并在每次文件修改时自增它，可以达成两个目的。

(1) 任何关于 service worker 文件的修改，即使是在缓存版本号中改变一位数字这样的微小变化，都可以让浏览器知道，是时候安装新的 service worker 来替代旧的激活 service worker 了。这将触发一个新的 `install` 事件，并导致新文件下载并存储到缓存中。
(2) 它为每一个版本的 service worker 都创建了一份单独的缓存（见图 4-5）。这一点很重要，因为即使我们已经更新了缓存，在用户关闭所有页面之前，旧的 service worker 依然是激活的。旧的 service worker 可能会用到缓存中的某些文件，而这些文件又是可以被新的 service worker 所修改的。通过让每个版本的 service worker 拥有自己的缓存，就可以确保不会出现意料之外的情况。

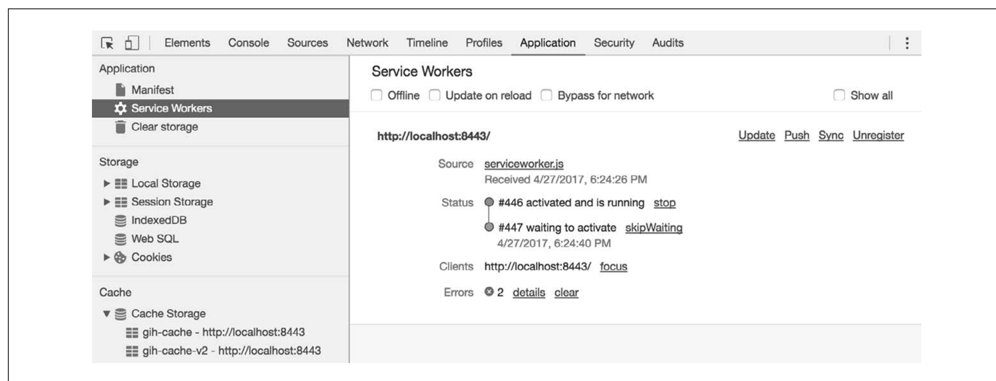


图 4-5：两个 service worker，各自拥有属于自己的缓存



我们选择了版本号作为缓存的名称（例如 `gih-cache-v2` 和 `gih-cache-v3`），这仅仅是为了开发的舒适性和可读性。浏览器并不知道 `gih-cache-v2` 比 `gih-cache-v3` 旧，只知道它们是不同的缓存。我们也可以简单地将其命名为 `cache-sierra` 和 `nevada-store`。

从技术上讲，`gih-cache-v2` 只是我们的缓存版本号名称，但我们还经常使用缓存名词指代 service worker 的版本。例如，我们可以把名为 `gih-cache-v2` 的缓存名用在 service worker 版本 2 中。一旦你决定为每个 service worker 版本维护一个单独的缓存版本，使用相同的名词来称呼它们可以让事情变得更简单。

4.5 缓存管理与清除旧缓存

通过给缓存和 service worker 添加版本，我们能够实现一套强大的系统，其中每个版本的 service worker 都可以只依赖于其专用缓存中的文件。我们可以随时更新 service worker 或者缓存中的文件，并确保不会以意想不到的方式影响到用户。别忘了，你可能刚刚把 service worker 的版本更新到 327，但是你的用户安装的 service worker 版本可能是 122。你真的需要确保你发布的每个 service worker 版本都要修改并保留一遍缓存文件吗？

这正好产生了一个问题。每次我们更新 service worker，都在用户设备上创建了一份新的缓存，把缓存弄得很杂乱。不仅如此，我们可能最终存储了 327 份标志图片，以及华丽的高分辨率封面图。不久之后，浏览器就会干涉我们，让我们知道，存储容量已经达到上限了。

存储限制

对于 CacheStorage 的管理，每个浏览器的行为都有所不同，包括如何给每个网站分配缓存空间，以及如何清除旧的缓存条目。不同的浏览器、浏览器版本号、设备，甚至随着时间推移（设备的剩余空间会发生变化），都会影响分配给你的站点的空间。

除了每个站点（即每个源）的存储限制之外，大多数浏览器还会设置一个所有缓存的大小限制。当缓存超出了这个限制之后，浏览器就会删除最久之前访问的网站缓存（也称为最近最少使用，least recently used）。

浏览器不会仅删除站点的部分缓存。要么删除你网站的所有缓存，要么不删除任何内容。这将确保你的站点不会处于一种不可预测的部分缓存状态。

我们的 service worker 要学会在浏览器中成为一名良好公民，不仅要创建缓存，还应该负责地处理不再需要使用的旧缓存资源。

在解决这个问题之前，我们需要熟悉 `caches` 对象的两个新方法。

`caches.delete(cacheName)`
: 接收一个缓存名字作为第一个参数，并删除对应的缓存。

`caches.keys()`
: 一个获取所有缓存名称的简便方法。返回一个 promise，其完成的时候会得到一个包含缓存名称的数组。

通过组合使用这两个方法，就可以创建代码来删除所有缓存或者某些缓存。例如，如果想删除所有缓存，可以使用下列代码：

```
caches.keys().then(function(cacheNames) {
  cacheNames.forEach(function(cacheName) {
    caches.delete(cacheName);
  });
});
```

接下来看看如何用此来管理缓存。



适用于这个应用的内容，未必适合你的现实情况。

虽然我决定将哥谭帝国酒店的所有静态资源存储在一份缓存中（对于每个版本），但是你可以针对你的应用定制一套不同的结构。举个例子，你可能会为一些不频繁更改的文件（例如第三方库、标志图案等）单独保留一份缓存，为那些随每次发布而修改的文件使用另一份缓存。如果确实如此，请务必修改此处所描述的模式。

任何时候，我们的应用最多需要两份缓存：一份用于当前激活的 service worker，另一份用于正在安装但尚未激活的 service worker（如果存在的话）。任何属于废弃 service worker 的缓存也是废弃的。

让我们将目标分解，并放进 service worker 生命周期中。

(1) 每次安装 service worker，我们都创建一份新的缓存。
(2) 当新的 service worker 激活的时候，就可以安全删除过去的 service worker 创建的所有缓存。

我们的代码已经完成了步骤 1，只需要添加步骤 2，我们的 service worker 就可以打扫自己家了。幸运的是，我们已经熟知了做这件事情的绝佳机会——`activate` 事件。

在现有 service worker 的基础上，我们添加一个新的事件监听器，监听 `activate` 事件。

在 serviceworker.js 中，把 `CACHE_NAME` 变量修改成 `gih-cache-v4`，然后在文件底部添加下列代码：

```
self.addEventListener("activate", function(event) {
  event.waitUntil(
    caches.keys().then(function(cacheNames) {
      return Promise.all(
        cacheNames.map(function(cacheName) {
          if (CACHE_NAME !== cacheName && cacheName.startsWith("gih-cache")) {
            return caches.delete(cacheName);
          }
        })
      );
    })
  );
});
```

我们的代码现在监听了另一个事件——`activate`。当一个已安装 / 等待中的 service worker 准备好激活，并替代旧的激活 service worker 的时候，就会调用这个事件。在这个阶段，它所需要的文件已经成功缓存了。但在我们宣称新的 service worker 激活之前，需要删除所有旧的缓存，这些缓存是旧的 service worker 用到的。

让我们逐行检查 `activate` 事件的代码。

首先我们使用 `waitUntil` 来扩展 `activate` 事件。本质上讲，我们让 service worker 完成激活之前，先等待我们删除所有的旧缓存。我们通过给 `waitUntil` 传入 promise 来实现这一点。

创建 promise 的代码一开始调用了 `caches.keys()`。这个方法返回一个 promise，当它完成的时候会提供一个数组，其中包含了我们在应用中创建的所有缓存的名称。我们需要拿到这个数组，然后创建一个 promise，只有完成迭代数组中的每个缓存，才能解决这个 promise。要实现这一点，我们可以使用 `Promise.all()` 把所有的 promsie 包裹在单个 promise 中。



`Promise.all()` 接收一个 promise 数组，并返回一个单独的 promise，一旦数组中的所有 promise 都完成，这个单独的 promise 也会完成。如果数组中的任何一个 promise 失败，`Promise.all()` 创建的 promise 也会失败。反之，如果所有 promise 都完成，这个单独的 promise 也会完成。

接下来，我们创建了一个 promise 数组，将其传递给 `Promise.all()`。我们是这样创建的：基于 `cacheNames` 数组，使用 `Array.map()` 方法，给每个缓存名称创建一个对应的 promise——这个 promise 会删除对应名称的缓存，然后完成。

要详细了解如何使用 `Array.map()` 来给每个缓存名称创建 promise，请参见后文的“为 `Promise.all()` 创建一个 promise 数组”。

拿到这个删除缓存的 promise 数组之后，我们将其传递给 `Promise.all()`，`Promise.all()` 转而返回一个单独的 promise，然后传给 `event.waitUntil()`。

现在我们还没介绍的唯一一行是 `if` 语句，其中包含了我们的 `caches.delete()` 调用。这个语句负责确保只会删除同时符合下列条件的缓存内容：

(1) 缓存名称不等于当前激活的缓存名称；
(2) 缓存名称以 `gih-cache` 开头。

第一个条件确保我们不会删除刚刚创建的新缓存。第二个条件则检查了所有 service worker 缓存名称的任意前缀。这确保了我们不会删除那些由应用其他地方创建的、与 service worker 无关的缓存。

尽管实现的事情很简单，但是由于这段代码使用了链式 promise 和一些不太常用的方法，看起来像是整本书中比较复杂的代码。我们通过伪代码来总结整个 `activate` 事件监听器，或者会更容易掌握：

```
监听activate事件：
  在声明service worker激活成功之前，需要等待下列内容完成
  如果下列所有事情都能成功完成：
    对于每个缓存名称：
      检查缓存名称，如果不等于当前缓存名
      并且名称以gih-cache开头：
        删除缓存。
```

为 `Promise.all()` 创建一个 promise 数组

当我们想要传递一个 promise，而这个 promise 等到所有其他 promise 全部完成的时候才会完成自身，这时可以使用 `Promise.all()` 方法。

`Promise.all()` 接收一个 promise 数组，并返回一个单独的 promise。这个 promise 仅在接收的所有 promise 全部完成的时候，才能完成自身。如果数组中的任何 promise 失败了，`Promise.all()` 返回的 promise 也会失败。

我们可以使用 `Array.map()` 来基于任何其他数组（例如上述例子中提到的缓存名称数组）创建对应的 promise 数组。我们可以通过给数组的 `map()` 方法传入一个回调函数来实现这一点，这个回调函数接收数组中的单个元素，并返回一个新的 promise。

下面的例子是一个简化的版本：

```
var values = [true, false, true, true];
Promise.all(
    values.map(function(val) {
```

```
        if (val === true) {
          return Promise.resolve();
        } else {
          return Promise.reject();
        }
      })
    )
    .then(function() {console.log("Everything is true");})
    .catch(function() {console.log("Not everything is true");});
```

这段代码为一个布尔值数组创建了一个新数组，其中包含了四个 promise，其中的三个会成功，但有一个将会失败。这将导致 Promise.all 返回的 promise 整个失败，并触发执行 catch 块的代码。

4.6 重用已缓存的响应

带版本号缓存实现，为我们提供了一个非常灵活的方式来控制缓存，并保持最新。但是，如果我们仔细研究它，可能会发现其内在实现是低效的。

每当我们创建一个新的缓存，会使用 cache.add() 或者 cache.addAll() 来缓存应用需要的所有文件。但是，如果用户已经在本地拥有了 cache-v1 缓存，而我们现在要创建 cache-v2，会发生什么呢？我们请求并放置到 cache-v2 中的某些文件，已经存在于 cache-v1 中。如果我们知道这些文件是永远不会改变的，就浪费了宝贵的带宽和时间来从网络上再次下载它们。

如果我们创建了一个新的缓存，首先遍历一份不可变文件的列表（其中包含了从不改变的文件，例如 bootstrap.3.7.7.min.css 或者 style-v355.css），然后从现有缓存中寻找它们，并直接复制到新的缓存中，会怎么样？完成这项工作后，我们就可以继续使用 cache.add() 或者 cache.addAll() 来获取剩下的文件（包括在旧缓存中找不到的不可变文件，以及可变的文件）。

```
var immutableRequests = [
  "/fancy_header_background.mp4",
  "/vendor/bootstrap/3.3.7/bootstrap.min.css",
  "/css/style-v355.css"
];
var mutableRequests = [
  "app-settings.json",
  "index.html"
];

self.addEventListener("install", function(event) {
  event.waitUntil(
    caches.open("cache-v2").then(function(cache) {
      var newImmutableRequests = [];
      return Promise.all(
        immutableRequests.map(function(url) {
          return caches.match(url).then(function(response) {
            if (response) {
              return cache.put(url, response);
            } else {
```

```
              newImmutableRequests.push(url);
              return Promise.resolve();
            }
          });
        })
      ).then(function() {
        return cache.addAll(newImmutableRequests.concat(mutableRequests));
      });
    })
  );
});
```

这段代码把需要的资源分成了两个数组。

(1) `immutableRequests` 中包含了我们知道永不改变的资源 URL。这些资源可以安全地在缓存之间复制。

(2) `mutableRequests` 中包含了每次创建新缓存时，我们都要从网络中请求的 URL。

首先，我们的 `install` 事件会遍历所有的 `immutableRequests`，并且在所有现有的缓存中寻找它们。寻找到的任何资源，都会使用 `cache.put` 复制到新的缓存中。[1] 而没有寻找到的资源，会被放入到 `newImmutableRequests` 数组中。

一旦所有的请求都检查完毕，代码就会使用 `cache.addAll()` 来缓存 `mutableRequests` 和 `newImmutableRequests` 中的所有 URL。



在大部分 service worker 中，这种模式都是实用的。为了减少你的代码输入量，我给 `cache.addAll()` 创建了一个替代方法，称为 `cache.adderall()`，简化了上述的模式。

```
importScripts("cache.adderall.js");

self.addEventListener("install", function(event) {
  event.waitUntil(
    caches.open("cache-v2").then(function(cache) {
      return adderall.addAll(cache, IMMUTABLE_URLS, MUTABLE_URLS)
    })
  )
});
```

你可以在线上找到关于 `cache.adderall()` 的更多信息。

4.7 配置服务器以提供正确的响应头部

由于 service worker 文件在每次加载的时候都会进行检查，你应该修改你的服务器配置，提供一个较短的过期时间头部（即 1 到 10 分钟）。如果你给它一个很长的过期时间，浏览器就不会检查它的变化，导致不能发现 service worker 的新版本或者需要缓存的新文件。

注 1：`cache.put` 接收一个键值对(例如 URL 作为键,response 对象作为值),然后在缓存中创建一个新的条目。与只需要提供 URL 的 `cache.add` 不同，`cache.put` 不会发起另一个网络请求，因为它已经包含了需要缓存的响应。

想象一下，如果你的 service worker 总是从缓存中提供 checkout.js，然后你意外地发布了一个有 bug 的文件版本，会发生什么。如果你不能通过更新 service worker 来缓存一个新版本的话，可能在数小时之内都不能修复你的应用。

幸运的是，浏览器有一项保护措施，默认的过期时间是 24 小时，如果你尝试设置更长的过期时间也不会生效。

4.8 开发者工具

当你了解本书中的 service worker、CacheStorage 以及其他新的 API 的时候，我鼓励你花时间学习不同浏览器中提供的开发者工具。

大部分的现代浏览器，例如 Chrome、Opera 和 Firefox，都提供了可以帮助你改进工作流程的工具，使代码开发和调试更加容易。

以下是我个人最常使用的一些开发者工具。

4.8.1 控制台

现代浏览器提供了令人惊叹的调试工具，包括设置断点、监听变量、跳入和跳出函数等。但也许是我年纪大了，没有耐心学习新技巧了，传统的控制台依然是我首选的调试工具。

使用 service worker 的时候，请记住，当你在任何选项卡中打开控制台的时候，命令都会运行在 `window` 上下文中，而不是 service worker。如果你想要探索 service worker 上下文，并在其中运行命令，就需要改变控制台的上下文。

在 Chrome 和 Opera 中，可以通过打开控制台，并将上下文从 `top`（也就是 `window`）修改成你的 service worker 文件来实现。选中的上下文在图 4-6 中以红色标记出来了。

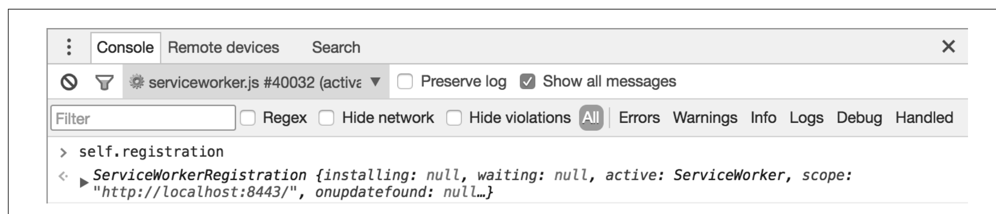


图 4-6：在 Chrome 中改变浏览器的上下文

在 Firefox 中，可以通过打开 `about:debugging#workers`，并点击 service worker 旁边的调试按钮实现同样的效果。如果你在 service worker 旁边没有看到调试按钮，可能需要先点击 `Start` 开始它。

4.8.2 清除缓存并刷新

在工作时，我们一直在修改代码和数据结构。在浏览器中查看应用的时候，我们通常想确保正在查看最新的代码、最近的数据，并且正在使用一份最新的缓存。

在过去，这很简单。只需要在刷新时按住 Shift 键，就可以实现硬性的重新加载，并确保忽略浏览器的缓存。但随着越来越多的静态资源存储在更多的地方（CacheStorage、IndexedDB、Cookies、Local Storage、Session Storage 等），仅仅这样做已经不够了。

幸运的是，Chrome 和 Opera 中的开发者工具提供了一个快速方法来实现清空。在其中一个浏览器的开发者工具中，打开 Application 标签页，选择 Clear storage 部分，勾选复选框，然后点击“Clear site data”即可。

4.8.3　检查CacheStorage和IndexedDB

在开发过程中，你经常需要检查存储在 CacheStorage 和 IndexedDB（参见第 6 章）中的资源。

你可以通过控制台编程访问这些存储资源，打开它们的连接，并读取其数据。但是这样相当麻烦。

Firefox、Chrome 和 Opera 都可以让你直接使用图形用户界面来检查这些存储资源。在 Firefox 中，可以通过开发者工具的 Storage 标签（见图 4-7）来访问。而在 Chrome 和 Opera 中，你可以通过 Application 标签来找到它。

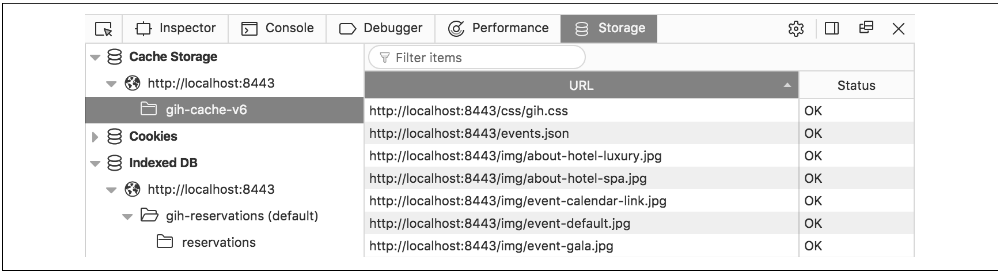


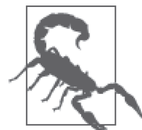
图 4-7：在 Firefox 中检查存储资源

4.8.4　网络节流与模拟离线情况

当我们在本地机器上开发应用时，总是在最佳的条件下看待工作，我们很容易遗忘一点，即这并不是用户将会真正体验到的应用。

当我们试图改进应用时，其中一个最有价值的工具就是模拟不同的连接速度以及模拟离线状态的能力。

在 Firefox 中，可以通过在开发者工具栏中打开响应式设计模式（Responsive Design Mode），然后修改窗口顶部的节流设置来实现连接节流。而在 Chrome 和 Opera 中，可以通过点击开发者工具的 Network 标签实现这一点，即点击节流控制，并选择不同的连接状况即可（如图 2-4 所示）。



记住，模拟移动设备的功能只能实现到以上程度。这不能替代实际设备上的真实测试。我强烈推荐通过 Alex Russel 的“Progressive Performance”演讲学习进行现实中的检查。

4.8.5 Lighthouse

Lighthouse 是一个开源工具，最初由 Google 开发，可用于自动审计你的应用是否符合一系列的 PWA 最佳实践。

Lighthouse 既可以通过浏览器扩展的方式运行（如图 4-8 所示），也可以通过命令行工具运行，你可以将其整合到你的持续集成流水线中。

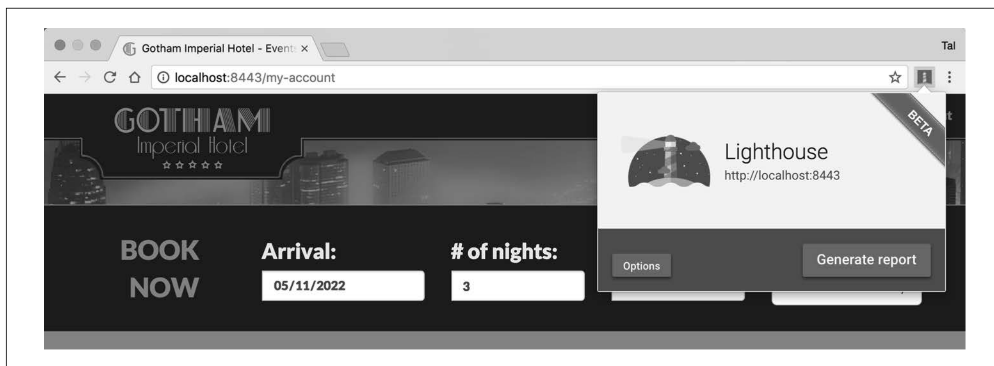


图 4-8：Lighthouse Chrome 浏览器扩展正在测试哥谭帝国酒店应用

4.9 小结

希望现在你更好地理解了 service worker 生命周期。

如果你仍然不确定为什么有时候在 service worker 开始工作之前需要刷新页面，或者为什么在更新 service worker 之后怎么刷新都不能让新代码工作，请考虑重新阅读一遍本章。

service worker 生命周期可能是其另一个会造成疑惑的方面。如果你忘记将其考虑进去，可能会花 20 分钟尝试调试一个特别讨厌的 bug，最后发现你的页面仍然被一个旧的 service worker 所控制。相信我，我在不久之前就经过过这种情况。

不过，理解 service worker 生命周期也为你提供了新的机会来完成一些令人惊奇的事情。通过理解 `install` 事件的原理，我们就可以为 service worker 创建安装依赖。通过理解 service worker 何时从已安装状态变成激活状态，只需几行代码，我们就能够创建一个可以管理多缓存版本的复杂系统。在接下来的几章中，我们将会发现更多的机会来运用这些知识去实现一些令人惊讶的事情。

第5章
拥抱离线优先

十多年前，我和其他人共同创办了一家名为 Wi-Ser 的公司。那时是 2004 年，我们的目标是，让咖啡馆和餐馆老板向他们的客户提供免费 WiFi。

在试图兜售这个革命性的想法时，我们遇到了两种主要的反对意见。

非技术怀疑者问我们："为什么我们需要向客户提供 WiFi？为什么他们在喝咖啡的时候需要上网？"

少数技术怀疑者指出，他们了解到 WiMAX 即将来临——有线连接很快将会被取代。

从那时起，十多年过去了，我们中的大多数人意识到，移动连接的问题不会很快"解决"。

坐在家里或者办公室里时，我们有一个可靠的互联网连接，所以很容易忽视这个问题。连接问题是世界角落中那些不幸的人面临的问题。但是，其实这个问题会影响我们所有人，无论是登机、在国外降落而又没有当地的数据流量、地铁通勤、徒步旅行，甚至只是坐在家中接收不到网络信号的房间，都不例外（除非站在一个相当高的位置）。

我们已经习惯了这一点，以至于"我要进入隧道了"或者"我要进电梯了"已经成为了笑话。当你想要中断和其他人的连接的时候，就会编造这样的借口。

在 Web 应用中，是时候停止把失去连接作为一种错误状态来处理了。离线和差连接都是应用中不可避免的状态，我们必须为此做打算。

当意识到不能再忽视用户通过移动端屏幕来访问网站的情况时，我们拥抱了移动优先原则。我们不得不接受这样一个事实：网站不能只适配 15 英寸的屏幕了。我们学会了优先考虑移动设备，并以此为基础构建用户体验。

然而，在日益增长的移动世界中，连接从未得到保证，带宽费用也非常高昂，我们也变得自满了。

随着我们的网站越来越复杂，变成了成熟的 Web 应用，网站的平均大小也在激增。很容易找到包含数兆字节内容的 Web 页面，尽管它们的内容是纯静态的。

世界已经转变成移动优先，但我们的连接和带宽思维却依然根植在桌面浏览器的时代。

是时候开始思考**离线优先**（offine-first）了。

5.1 什么是离线优先

传统 Web 应用完全依赖于服务器。所有的数据、内容、设计和应用逻辑都存储在服务端上。客户端仅仅用来将一些 HTML 内容渲染到屏幕上。但随着 Web 应用的发展，越来越多的逻辑和能力转移到了客户端。Web 应用开始进行数据处理、模板渲染等工作。但是，和原生应用不一样的是，我们的 Web 应用依然完全依赖于服务器。任何连接的中断都会导致应用完全崩溃。

离线优先接受了一个简单的事实：离线和低连接的情况是不可避免的。这些情况不应该被视为灾难性的失败，而只是 Web 应用生命中另一种可能的状态。这是一种你应该规划在内并优雅处理的状态。

拥抱离线优先意味着接受这一点：尽管应用的某些功能在用户离线时可能不能正常使用，但更多的功能应该保持可用。



让我们回顾 2.2 节的“挑战不同，实现方式各异”中的示例消息应用。传统上，如果用户在离线时访问这个 Web 应用，浏览器只会显示一个错误。但是在原生应用中，这种糟糕的用户体验是不可接受的。我们没有理由不让 Web 应用达到同样的用户体验标准。

现代的消息 PWA 可以在本地缓存其接口和逻辑，以及最近的消息内容。然后，它可以将最后一次缓存的内容以及完整的用户界面展示给用户。虽然内容可能有点过时（和用户进行沟通也很重要），但它仍然是有用的。这个应用将不再把失去网络连接作为灾难性的失败处理。它为用户提供了当前条件下可能的最佳体验。

离线优先的另外一个基本方面是优雅地处理这些连接的变化。优雅地处理丢失的连接，意味着向用户传达某些功能可能不可用，或者他正在查看的数据可能是几小时之前的，但仍然尽可能多地暴露功能。即使你已经构建了一个完全离线可用的 Web 应用，优雅地处理连接变化也意味着用户可以放心使用这个应用，并且他的数据不会丢失。



回到消息应用的示例中，开发者还需要决定如何处理用户在离线状态下发送的新消息。应用可以“优雅地禁用”输入框，让用户知道不能在离线状态下发送消息。或者也可以让用户输入新消息，保存在浏览器的本地数据库中（参见第 6 章），然后在连接重新建立的时候立即发送。应用甚至可以在发送消息时（无论用户离线还是在线）使用后台同步（参见第 7 章）来保证信息总能发送，不管连接如何变化。

移动优先意味着：总是基于用户设备，提供最佳体验。

离线优先意味着：总是基于当前网络条件，提供最佳体验。

5.2 常用缓存模式

在本章结尾，我们将让哥谭帝国酒店网站完全符合离线优先原则。

在为站点不同部分制定缓存策略之前，我们需要熟悉一些用于缓存的常见设计模式。

不同的模式适合不同的情况，大多数应用都会使用几种不同的模式。例如，如果我们要创建天气应用，可能希望采用的模式是：总是加载来自网络的最新天气数据，并且只在网络请求失败时才尝试从缓存中获取。另外，对于展示不同天气情况的图标，我们可能倾向于采用另一种模式：总是首先从缓存中获取图标，只有在缓存中找不到的时候，才尝试去请求网络。

天气状况是资源迅速变化的一个例子，其关键是要展示最新的数据。至于描绘局部多云的图标，既不受时间影响，也不会经常变化。

我们来探讨一些更常见的缓存模式。[1]

仅缓存

从缓存中响应所有的资源请求。如果在缓存中找不到，请求会失败。该模式假定资源以前缓存过，最有可能用作 service worker 安装期间的依赖项。

这对于静态资源是实用的，因为静态资源不会在发布之间发生变化，例如徽标、图标和样式表。这并不意味着你永远无法修改它们，只是意味着它们不会在应用某个特定版本的生命周期内发生变化。

如果这些文件确实发生了变化，可以通过重新命名并将这些新文件存储到缓存中来更新它们。这类似于传统的缓存实践（和 service worker 无关），即在每个版本（例如 style.v1.0.3.css 或者 main_ae3f7.js）中，修改所有静态文件的名称，并且配置服务器，使得在提供这些文件时，携带一个非常长（甚至是无限长）的缓存过期时间。

如果选择不修改文件名，可以通过发布 service worker 的一个新版本再次获取这些文件，然后在 service worker 的激活事件中进行缓存（参见第 4 章）。

```
self.addEventListener("fetch", function(event) {
  event.respondWith(
    caches.match(event.request)
  );
});
```

缓存优先，网络作为回退方案

和仅缓存类似，这个模式也会从缓存中响应请求。然而，如果在缓存中找不到内容，service worker 会尝试从网络中请求并返回：

注 1：这些模式以及一些其他内容，首先是在 Jake Archibald 的 *The Offline Cookbook* 中被分类和命名的。强烈推荐这本书。

```
self.addEventListener("fetch", function(event) {
  event.respondWith(
    caches.match(event.request).then(function(response) {
      return response || fetch(event.request);
    })
  );
});
```

仅网络

经典的 Web 模型。尝试从网络中请求。网络不通，则请求失败。可以用于不缓存的内容，例如用于数据统计的请求。

你很少需要使用这种模式，因为只需要忽略 service worker 的 fetch 事件，让其默认行为发挥作用即可实现仅网络请求。但是，如果你发现自己需要以编程方式执行这种网络请求，下面的代码可能会有帮助。

```
self.addEventListener("fetch", function(event) {
  event.respondWith(
    fetch(event.request)
  );
});
```

网络优先，缓存作为回退方案

总是向网络发起请求。请求失败则返回缓存中的版本。如果在缓存中找不到，请求就会失败。

```
self.addEventListener("fetch", function(event) {
  event.respondWith(
    fetch(event.request).catch(function() {
      return caches.match(event.request);
    })
  );
});
```

用户总能获取到当前连接状况下可用的最新内容。对于经常改变的内容来说这很合适，对于需要显示最新响应的场景来说也很重要。

先缓存，后网络

在检查网络是否有较新版本的期间，立即显示缓存中的数据。一旦网络返回响应，检查它是否比缓存新，并使用新内容更新页面。

虽然这可能看起来是一种最好的方法，将缓存的快速响应和网络中可用的最新内容结合了起来，但是它需要付出代价。

你必须修改你的应用才能发起两次请求，先显示缓存内容，最后在新内容可用的时候更新页面。更重要的是，这种模式可能会为你的应用带来新的 UX（用户体验）挑战。虽然把一张图片替换成可用的新内容很容易，但是如果你要更新的内容是用户正在编辑的文档文本呢？如果用户已经开始编辑第二句话，你又需要修改它，应该如何处理呢？修改的最佳方式是什么，你又应该如何与用户沟通这个修改呢？

通用回退

当用户请求的内容在缓存中找不到，并且网络不可用时，该模式从缓存中返回一个替代的“默认回退”版本，而不是返回一个错误。

一种常见的用法是返回一张通用图像，代替某个特定图像。例如，当用户头像在缓存中找不到，网络也不可用的时候，可以显示一张通用的头像，而不是在应用中留下一张损坏的图像。这种方法非常棒，而且可以在连接状态变化的时候优雅地进行处理。

这种模式通常会与其他模式一起使用，作为最终的回退方案。下面的示例演示了它如何与**网络优先，缓存作为回退方案**模式一起使用，创建一种**网络优先，缓存作为回退方案，通用回退作为兜底方案**的模式：

```
self.addEventListener("fetch", function(event) {
  event.respondWith(
    fetch(event.request).catch(function() {
      return caches.match(event.request).then(function(response) {
        return response || caches.match("/generic.png");
      });
    })
  );
});
```



Twitter 使用 PWA 敲开新兴市场的大门

对于 Twitter 来说，渐进式 Web 应用是个福音——尤其是在进入新兴市场的路上，仍然有着大量的增长机会。这些市场往往具有昂贵、缓慢、不可靠连接等特征。

Twitter 的渐进式 Web 应用结合了他们原生应用中的许多好处。其大小仅为 400KB（大约是安卓原生版本大小的 2.5%），使用的电量更少，比原生应用从首页启动的时间还快了几秒钟。更重要的是，它不仅结合了所有这些好处，还没有牺牲本地应用的任何特性。

所有的这些好处给 Twitter 带来了显著的优势。在市场上，较慢的功能手机，以及不可靠又昂贵的连接才是常态。

5.3　混合与匹配：创造新模式

我们已经看到了一些更常见的缓存模式，让我们研究一下如何将它们组合起来，以创建缓存和服务内容的新方法。

按需缓存

对于不经常改变的资源以及 service worker install 事件期间不想缓存的资源，我们可以扩展**缓存优先，网络作为回退方案**模式，将从网络返回的请求保存到缓存中。

这样可以有效地创建一个按需缓存资源的系统。资源被第一次请求时，在缓存中是找不到的。service worker 将从网络中检索资源，保存到缓存中，然后返回。当下次再请求该资源时，它将立即从缓存中返回。

```
self.addEventListener("fetch", function(event) {
  event.respondWith(
    caches.open("cache-name").then(function(cache) {
      return cache.match(event.request).then(function(cachedResponse) {
        return cachedResponse || fetch(event.request).then(
          function(networkResponse) {
          cache.put(event.request, networkResponse.clone());
          return networkResponse;
        });
      })
    })
  );
});
```



克隆示例——多次使用同一响应

当你查看上述模式的代码时，可能已经注意到一点新的内容。在将响应保存到缓存中时，我们对其调用了一个 clone 方法：

```
fetch(request).then(function(response) {
  cache.put(request, response.clone());
  return response;
});
```

为什么我们调用 put 放入缓存的是一份克隆副本，而不是响应本身呢？

实际上，这和我们使用 cache.put() 并没有什么关系。之前我们放入缓存的响应并没有先进行克隆。真正的原因是，我们打算不止一次地使用这个响应。

你可以这样理解：响应是写在一张纸上的。假如哥谭帝国酒店的主人，也就是德维恩家族财富的继承人，准备了一次演讲并将其写在了一张纸上。就在他准备要上台的时候，他把演讲稿交给助手存放。一旦他登台，可能会发现自己站在一个拥挤的房间前面，手上什么也没有——除非他先把演讲稿抄了一份，然后再交给他的助手。

响应也是同样的道理。你可以将它从一个位置传递到另一个位置（例如使用 return 语句），但是如果你打算不止一次地使用它（例如，将其放入缓存并使用它来响应事件），请确保使用 clone 命令来复制它。

缓存优先，网络作为回退方案，并频繁更新缓存

对于经常改变的资源，如果显示最新版本的优先级不如返回快速响应（例如用户头像），我们可以修改**缓存优先，网络作为回退方案**模式，即使在缓存中可以找到，也总会从网络请求资源。这种模式从缓存中快速响应，同时获取更新的版本并在后台缓存。在用户下次请求该资源的时候，从网络中请求的资源导致的任何变化都将生效。这种模式将快速响应和相对较新的响应（显示上次请求时的最新内容）相结合。

```
self.addEventListener("fetch", function(event) {
  event.respondWith(
    caches.open("cache-name").then(function(cache) {
      return cache.match(event.request).then(function(cachedResponse) {
```

```
        var fetchPromise =
          fetch(event.request).then(function(networkResponse) {
            cache.put(event.request, networkResponse.clone());
            return networkResponse;
          });
        return cachedResponse || fetchPromise;
      })
    })
  );
});
```

我们的事件处理器以 `event.respondWith` 开始，余下的代码都在构建这个响应。

首先我们打开一个缓存，并试图在其中寻找匹配的请求。不管是否匹配成功，`cache.match` 返回的 promise 都会成功，随后调用 `then` 回调函数。回调函数一开始会创建一个新的 `fetch` 请求，用于请求资源，保存到缓存中，并返回响应。代码的最后一行返回给 `event.respondWith` 的要么是已缓存的响应，要么是在没有找到缓存的请求下返回网络响应的 promise。

当我们调用 `fetch` 的时候，会返回一个 promise，并继续执行脚本，同时 `fetch` 操作是异步完成的。这允许我们在不等待 `fetch` 完成的情况下，直接返回 `cachedResponse`，或者返回 `fetch` 创建的 promise（这个 promise 在完成时会带有网络返回的文件）。

网络优先，缓存作为回退方案，并频繁更新缓存

如果“始终提供可用资源的最新版本”很重要，可以对**网络优先，缓存作为回退方案**稍作修改。和原始的模式类似，该模式总会试图从网络中获取最新版本，仅在网络请求失败的时候才回退到缓存版本。此外，每当网络成功访问时，会将当前缓存更新为网络响应的内容。

```
self.addEventListener("fetch", function(event) {
  event.respondWith(
    caches.open("cache-name").then(function(cache) {
      return fetch(event.request).then(function(networkResponse) {
        cache.put(event.request, networkResponse.clone());
        return networkResponse;
      }).catch(function() {
        return caches.match(event.request);
      });
    })
  );
});
```

5.4 规划缓存策略

直到目前为止，我们在哥谭帝国酒店应用中处理连接问题的方法完全是基于**网络优先，缓存作为回退方案**模式。我们使用这个模式缓存了首页的一个简化版本，并在检测到网络错误时提供给用户。

这已经比我们原本的应用有了显著的改进。我们知道在加载了这一功能后，可以为用户提供附加的价值。

现在，我们已经了解了不同的缓存模式，可以再进一步了。

是时候结合目前所学到的知识，采用离线优先的方式构建哥谭帝国酒店应用了。当我们完成时，页面本身会即时加载，其中那些随时间改变的资源将会从网络加载，如果网络不可用，则从缓存中加载。

让我们检查一下首页。

首页由静态的 index.html 文件组成，它很少会随着版本发生变化。它会请求多个静态图片、样式表和 JavaScript 文件。index.html 使用的所有静态文件都可以在安装过程缓存下来，并且完美适合**缓存优先，网络作为回退方案**模式。这将为用户提供更加快速的加载时间，无论用户是在线、离线还是介于两者之间。

至于 index.html 文件本身呢？由于这个文件很少在版本之间变化，所以我们可能会想到**缓存优先，网络作为回退方案**模式。但是，在这个场景下，这样的做法确实会带来很大的负面作用。如果这个文件更新，我们不得不同时更新 service worker，以确保获取和缓存新的文件。

更糟的是，在旧的 service worker 释放页面控制，新的 service worker 激活之前，用户都不会看到新的版本。在表 5-1 中，你可以看到每次访问的情况。

表5-1：每次访问对应的页面和service worker状态

访次	说明	service worker	index.html
1		安装 SW v1，并缓存 HTML v1	HTML v1 从网络提供
2	新版 service worker（v2）和新版 HTML（v2）可用	安装 SW v2，并缓存 HTML v2。SW v1 依然在控制页面	HTML v2 从缓存提供
3	SW v1 有机会释放页面控制	SW v2 激活，并控制页面	HTML v2 从网络提供

这意味着即使我们用新的 HTML 文件更新 service worker，它也不会在用户下次访问应用的时候显示。

第 4 章详细说明了为何会发生这种情况，以及 service worker 是如何在这些状态之间变化的。

我们来考虑缓存 index.html 的可选方案。

(1) 使用**缓存优先，网络作为回退方案**模式提供服务。其缺点是有可能不会显示可用的最新版本，即使它可能已经被缓存。然而，这是一种快速、节省带宽的方案。

(2) 使用**网络优先，缓存作为回退方案**模式提供服务。这样将会总是展示最新的文件。其缺点是我们错过了改善 HTML 文件加载时间的机会，因为 HTML 文件可能已经存在于缓存中。

(3) 使用**缓存优先，网络作为回退方案，并频繁更新缓存**模式。和方案 1 类似，该方案总是从缓存中提供 index.html，并提供非常快的响应时间。此外，它还会检查 index.html 文件的更新，如果存在则更新缓存，而不需要我们更新 service worker 的版本。下次用户加载页面时，就能看到最新的文件。这种方式将快速响应时间和几乎总是最新的文件结合起来。然而，它使用的带宽和方案 2 一样多，或者说就像根本没有使用 service worker 一样。

对于哥谭帝国酒店的首页，我选择使用方案 3。无论用户连接状况如何，这种方式都可以即时加载首页。这样做的主要缺点是，有时可能显示的首页是旧的，直到用户刷新页面（在这种情况下问题不大，因为所有可能变化的动态数据都是使用提供最新数据的方式缓存的）。第二个缺点是，每次访问都要从网络中获取 HTML，即使它可能已经在缓存中（同样，对于较小的文件来说这个问题不大，服务器可以发送 `Expires` 和 `ETag` 头部，确保它可以缓存在 HTTP 缓存中）。

继续往下看我们的首页，可以看到一个使用谷歌地图 JavaScript API 创建的地图。每次加载页面时，这个交互式地图会从谷歌服务器进行加载。由于不能缓存谷歌地图的所有逻辑和数据，我们可以更新 service worker，当检测到谷歌地图 JavaScript 文件加载失败时，提供一个替代的 JavaScript 文件。这个文件会显示一张静态的地图图片，而不是动态的、交互式的地图控件。这就是渐进增强的行为。离线用户看到的地图是静态图片，而在线用户将获得完全交互式的地图。

我们的首页还加载了一个 JSON 文件，其中包含即将在酒店发生的事件列表。这是随时可以更改的数据，我们希望用户总是能够看到最新的版本。我们将使用**网络优先，缓存作为回退方案，并频繁更新缓存**模式来提供这个文件。这样可以确保我们总是能够根据网络状况来提供最新数据：如果用户在线，则显示实时数据，否则显示缓存中的最新版本。

在事件 JSON 文件中还包括了多张图片文件的引用，每张图片分别表示一个不同的事件。由于这些文件在安装阶段没有被缓存，我们可以使用**按需缓存**模式。每次请求这些文件中的任何一个时，我们都会尝试从缓存中请求。如果在缓存中没有找到，就向网络发起请求，然后将网络返回的内容存储到缓存中以备下次使用，并返回给页面。

要确保页面上不会显示损坏的图像，我们还要修改图像事件的缓存代码，以便在缓存中找不到图像，并且在网络不可用时显示一张默认的回退图像。在安装阶段，默认图像会作为安装依赖项被缓存起来。

最后，我们要建立一项规则，确保数据统计请求直接发送到网络，而且不需要任何缓存或者回退。如果用户脱机，这些请求应该会失败。

让我们总结一下首页的缓存策略。

(1) 使用**缓存优先，网络作为回退方案，并频繁更新缓存**模式返回 index.html 文件。
(2) 使用**缓存优先，网络作为回退方案**模式返回首页需要展示的所有静态文件。
(3) 从网络中返回谷歌地图的 JavaScript 文件。如果请求失败，返回一个替代的脚本。
(4) 使用**网络优先，缓存作为回退方案，并频繁更新缓存**模式，返回 events.json 文件。
(5) 使用**按需缓存**模式返回事件的图片文件，如果网络不可用并且图片没有缓存，则回退到默认的通用图片。
(6) 数据分析的请求直接通过，不作处理。

5.5　实现缓存策略

在开始之前，请通过在命令行中运行下列命令，确保你的代码处于第 4 章结束时的状态。

```
git reset --hard
git checkout ch05-start
```

现在，我们通过更新 service worker，让其缓存并提供整个首页，以及首页渲染所需的所有静态资源，来开始实现新的缓存策略了。

将 serviceworker.js 中的代码替换成下列代码：

```
var CACHE_NAME = "gih-cache-v4";
var CACHED_URLS = [
  // HTML
  "/index.html",
  // 样式表
  "/css/gih.css",
  "https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css",
  "https://fonts.googleapis.com/css?family=Lato:300,600,900",
  // JavaScript
  "https://code.jquery.com/jquery-3.0.0.min.js",
  "/js/app.js",
  // 图片
  "/img/logo.png",
  "/img/logo-header.png",
  "/img/event-calendar-link.jpg",
  "/img/switch.png",
  "/img/logo-top-background.png",
  "/img/jumbo-background.jpg",
  "/img/reservation-gih.jpg",
  "/img/about-hotel-spa.jpg",
  "/img/about-hotel-luxury.jpg"
];

self.addEventListener("install", function(event) {
  event.waitUntil(
    caches.open(CACHE_NAME).then(function(cache) {
      return cache.addAll(CACHED_URLS);
    })
  );
});

self.addEventListener("fetch", function(event) {
  event.respondWith(
    fetch(event.request).catch(function() {
      return caches.match(event.request).then(function(response) {
        if (response) {
          return response;
        } else if (event.request.headers.get("accept").includes("text/html")) {
          return caches.match("/index.html");
        }
      });
    })
  );
});

self.addEventListener("activate", function(event) {
  event.waitUntil(
```

```
    caches.keys().then(function(cacheNames) {
      return Promise.all(
        cacheNames.map(function(cacheName) {
          if (CACHE_NAME !== cacheName && cacheName.startsWith("gih-cache")) {
            return caches.delete(cacheName);
          }
        })
      );
    })
  );
});
```

这段代码和第 4 章结尾的代码非常相似，除了两个地方有所区别。

首先，我们替换了 CACHED_URLS 数组的内容，把其中的 sw-index.html 换成 index.html，以及显示它所需要的所有静态文件。

第二处修改是在 fetch 监听器中，现在从缓存中返回的是 index.html 而不是 sw-index.html。

这两处小变化足以让离线用户的首页几乎和在线用户的保持一致。

让我们进一步研究并改进它，使得即使用户在线，也能做到瞬间加载 index.html 和它需要的所有静态文件。

在 serviceworker.js 中，把 fetch 事件监听器的代码替换成下列代码：

```
self.addEventListener("fetch", function(event) {
  var requestURL = new URL(event.request.url);
  if (requestURL.pathname === "/" || requestURL.pathname === "/index.html") {
    event.respondWith(
      caches.open(CACHE_NAME).then(function(cache) {
        return cache.match("/index.html").then(function(cachedResponse) {
          varfetchPromise =
            fetch("/index.html")
            .then(function(networkResponse) {
              cache.put("/index.html", networkResponse.clone());
              return networkResponse;
            });
          return cachedResponse || fetchPromise;
        });
      })
    );
  } elseif (
    CACHED_URLS.includes(requestURL.href) ||
    CACHED_URLS.includes(requestURL.pathname)
  ) {
    event.respondWith(
      caches.open(CACHE_NAME).then(function(cache) {
        return cache.match(event.request).then(function(response) {
          return response || fetch(event.request);
        });
      })
    );
  }
});
```

现在，新的 fetch 事件处理器会根据每个请求的 URL 表现出不同的行为。

我们首先检测的情况是请求根域名或者 /index.html（两种方式都可以请求首页）。我们使用**缓存优先，网络作为回退方案，并频繁更新缓存**模式处理这个请求。代码会在缓存中查找 index.html，无论是否找到，代码都会开始向网络请求并缓存最新版本。随后，要么立即返回缓存的版本，要么在缓存中找不到的情况下返回一个 promise，并等待其返回网络响应。

由于 fetch 是异步运行的，在 fetch 完成之前就可以返回缓存中的响应。

这个模式既可以让我们从缓存中得到即时响应（几毫秒内），同时又保证了 HTML 文件相对较新。

5.2 节中解释了这段代码的细节。

在事件监听器的结尾，我们检测了请求是否匹配 service worker 安装过程中缓存的 URL。如果匹配，我们使用缓存响应事件。如果在缓存中没有找到，我们就尝试从网络返回（这就是**缓存优先，网络作为回退方案**模式）。

如果请求没有匹配这两个条件中的任何一个，我们让其简单地通过 service worker，正常处理。



new URL(urlString, [baseURL])

在 fetch 事件监听器中，主要的条件语句通过检测 URL 来决定如何处理不同的请求。在过去，这需要通过一些相当烦人的正则表达式来完成。幸运的是，相对较新的 URL 接口让我们可以轻松完成这一点：

```
// 以下三个语句会返回相同的URL
var url_1 = new URL("https://gothamimperial.com/index.html");
var url_2 = new URL("/index.html", "https://gothamimperial.com");
var url_3 = new URL("/index.html", url_1);

// 下列所有的语句都为true
url_1.href     === "https://gothamimperial.com/index.html";
url_1.protocol === "https:";
url_1.hostname === "gothamimperial.com";
url_1.pathname === "/index.html";
```

我们刚刚完成了在 5.4 节中设定的第一个和第二个缓存目标。让我们再给事件处理器增加几个条件，来为不同的资源进行自定义。

把 serviceworker.js 的代码替换成下列代码：

```
var CACHE_NAME = "gih-cache-v5";
var CACHED_URLS = [
  // HTML
  "/index.html",
  // 样式表
  "/css/gih.css",
  "https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css",
  "https://fonts.googleapis.com/css?family=Lato:300,600,900",
  // JavaScript
  "https://code.jquery.com/jquery-3.0.0.min.js", "/js/app.js",
  "/js/offline-map.js",
  // 图片
```

```
  "/img/logo.png",
  "/img/logo-header.png",
  "/img/event-calendar-link.jpg",
  "/img/switch.png",
  "/img/logo-top-background.png",
  "/img/jumbo-background-sm.jpg",
  "/img/jumbo-background.jpg",
  "/img/reservation-gih.jpg",
  "/img/about-hotel-spa.jpg",
  "/img/about-hotel-luxury.jpg",
  "/img/event-default.jpg",
  "/img/map-offline.jpg",
  // JSON
  "/events.json"
];
var googleMapsAPIJS = "https://maps.googleapis.com/maps/api/js?key="+
  "AIzaSyDm9jndhfbcWByQnrivoaWAEQA8jy3COdE&callback=initMap";

self.addEventListener("install", function(event) {
  event.waitUntil(
    caches.open(CACHE_NAME).then(function(cache) {
      return cache.addAll(CACHED_URLS);
    })
  );
});

self.addEventListener("fetch", function(event) {
  var requestURL = new URL(event.request.url);
  // 处理index.html的请求
  if (requestURL.pathname === "/" || requestURL.pathname === "/index.html") {
    event.respondWith(
      caches.open(CACHE_NAME).then(function(cache) {
        return cache.match("/index.html").then(function(cachedResponse) {
          var fetchPromise = fetch("/index.html")
            .then(function(networkResponse) {
              cache.put("/index.html", networkResponse.clone());
              return networkResponse;
            });
          return cachedResponse || fetchPromise;
        });
      })
    );
  // 处理谷歌地图JavaScript API文件
  } else if (requestURL.href === googleMapsAPIJS) {
    event.respondWith(
      fetch(
        googleMapsAPIJS+"&"+Date.now(),
        { mode: "no-cors", cache: "no-store" }
      ).catch(function() {
        return caches.match("/js/offline-map.js");
      })
    );
  // 处理事件JSON文件请求
  } else if (requestURL.pathname === "/events.json") {
    event.respondWith(
```

```
      caches.open(CACHE_NAME).then(function(cache) {
        return fetch(event.request).then(function(networkResponse) {
          cache.put(event.request, networkResponse.clone());
          return networkResponse;
        }).catch(function() {
          return caches.match(event.request);
        });
      })
    );
  // 处理事件图片请求
  } else if (requestURL.pathname.startsWith("/img/event-")) {
    event.respondWith(
      caches.open(CACHE_NAME).then(function(cache) {
        return cache.match(event.request).then(function(cacheResponse) {
          return cacheResponse ||
            fetch(event.request).then(function(networkResponse) {
              cache.put(event.request, networkResponse.clone());
              return networkResponse;
            }).catch(function() {
              return cache.match("/img/event-default.jpg");
            });
        });
      })
    );
  // 处理统计请求
  } else if (requestURL.host === "www.google-analytics.com") {
    event.respondWith(fetch(event.request));
  // 处理在安装阶段已经缓存的请求
  } else if (
    CACHED_URLS.includes(requestURL.href) ||
    CACHED_URLS.includes(requestURL.pathname)
  ) {
    event.respondWith(
      caches.open(CACHE_NAME).then(function(cache) {
        return cache.match(event.request).then(function(response) {
          return response || fetch(event.request);
        });
      })
    );
  }
});

self.addEventListener("activate", function(event) {
  event.waitUntil(
    caches.keys().then(function(cacheNames) {
      return Promise.all(
        cacheNames.map(function(cacheName) {
          if (CACHE_NAME !== cacheName && cacheName.startsWith("gih-cache")) {
            return caches.delete(cacheName);
          }
        })
      );
    })
  );
});
```

这段代码示例引入了一些修改。

首先，它添加一些新文件到 CACHED_URLS 数组中（包括 /js/offline-map.js、/img/event-default.jpg、/img/map-offline.jpg 和 /events.json）。接下来，设置一个新的 googleMapsAPIJS 变量，其中包含了我们需要调用的谷歌地图 API 的 URL 地址（在这里设置一次是为了避免后续重复）。最后，它在 fetch 事件监听器中加入了一些条件判断。

第一个和最后一个条件保持不变。在这两个条件之间增加了四个新条件。我们逐一来看。

第一个新的条件是寻找谷歌地图 JavaScript API 请求：

```
if (requestURL.href === googleMapsAPIJS) {
  event.respondWith(
    fetch(
      googleMapsAPIJS+"&"+Date.now(),
      { mode: "no-cors", cache: "no-store" }
    ).catch(function() {
      return caches.match("/js/offline-map.js");
    })
  );
}
```

如果当前请求的是谷歌地图 JavaScript 文件，我们会尝试从 Web 获取。如果用户离线，请求就会失败，我们从缓存返回一个替代的 JavaScript 文件。这个简单的 JavaScript 文件（称为 offline-map.js）只包含了一行代码：

```
document.getElementById("map-container").classList.add("offline-map");
```

如果用户离线，这段代码会取代谷歌地图 API 的代码并运行，并添加一个名为 offline-map 的类名到 map-container div 中。如果你检查 CSS 文件就会发现，这个类负责把 div 的背景图片设置为地图的静态图。

请注意，我们同时还将静态图和新的 JavaScript 文件添加到 CACHED_ARRAY 数组中，以确保在 service worker 安装的时候，这两个文件都会被缓存。

对于我们的离线地图代码，最后还有两点需要注意。第一，在请求谷歌地图 JavaScript 文件时，我们需要通过 no-cors 模式进行请求，否则谷歌的服务器会拒绝我们的请求（参见附录 C）。第二，由于谷歌服务器返回的地图 API JavaScript 文件中包含的头部信息会导致浏览器总是试图从 HTTP 缓存中返回，因此我们需要确保它总是从网络中请求。否则我们的请求操作不会失败，就会导致谷歌地图一直控制页面（从缓存中），却不能加载地图（因为地图数据没有被缓存）。在请求时，通过把 cache 选项设置成 no-store 就可以完全跳过缓存，以实现这一目标。不幸的是，在本书编写时，并不是所有浏览器都支持这一选项，因此我们还需在每次请求时，往查询字符串添加了一个去缓存用的时间戳，以确保每次请求都是独立的，并忽略缓存。只需要把当前时间添加到每次请求的 URL 后面，即可实现这一点。

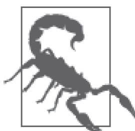
在 fetch 事件处理器中，第二个新的条件处理了包含事件数据的 JSON 文件请求：

```
if (requestURL.pathname === "/events.json") {
  event.respondWith(
```

```
    caches.open(CACHE_NAME).then(function(cache) {
      return fetch(event.request).then(function(networkResponse) {
        cache.put(event.request, networkResponse.clone());
        return networkResponse;
      }).catch(function() {
        return caches.match(event.request);
      });
    })
  );
}
```

由于数据经常变化，我们希望总能提供当前能访问到的最新数据，所以选择了**网络优先，缓存作为回退方案，并频繁更新缓存**模式。

首先，我们打开了缓存（无论网络请求是否成功，都需要进行这一步）。然后尝试向网络发起请求。如果请求成功，将响应放入缓存并返回。否则，我们会查找缓存的响应作为代替。



如果你没有跳过第 4 章，可能会注意到这里存在一个问题。我们只有在拦截 `fetch` 请求的时候，才缓存 events.json 文件。这只能在 service worker 控制页面之后才能发生。换句话说，在用户第一次访问页面时，这个文件是不会缓存的。只有当用户第二次访问时，service worker 才能捕获到浏览器尝试请求这个文件。如果用户在第二次访问时已经离线，在缓存中就不会找到这个文件。

由于 service worker 依赖这个文件存在于缓存中，因此我们可以通过把 events.json 添加到 `CACHED_URLS` 数组中，解决这个问题。这样就可以确保在 service worker 安装时将其缓存下来。随后，通过刚才添加的代码，就可以在每次连续访问时，保持其内容不断更新。

我们继续来看处理事件图片请求的条件：

```
if (requestURL.pathname.startsWith("/img/event-")) {
  event.respondWith(
    caches.open(CACHE_NAME).then(function(cache) {
      return cache.match(event.request).then(function(cacheResponse) {
        return cacheResponse ||
          fetch(event.request).then(function(networkResponse) {
            cache.put(event.request, networkResponse.clone());
            return networkResponse;
          }).catch(function() {
            return cache.match("/img/event-default.jpg");
          });
      });
    })
  );
}
```

由于这些图片经常发生变化，在开发过程中，我们无法获知客户在酒店里将要举办什么事件，所以我们将按需缓存这些资源。

每当我们检测到事件图片的请求时，一开始会打开缓存并试图寻找。随后，可能返回在缓存中找到的图片，也可能会尝试从网络中获取。如果图片请求成功，我们就将其放入缓

存，以备将来使用并返回它。



如果用户在第二次访问页面时离线，也会导致上一个条件中的类似问题。在这种情况下，用户会缓存 events.json 文件并尝试显示事件图片。不幸的是，图片还没有被缓存，也不能向网络请求。要处理这种边界情况，可以使用通用回退模式。如果图片没有缓存且无法从网络中请求，我们就回退到一张通用的事件图片（见图 5-1）。

别忘了把回退图片（`/img/event-default.jpg`）添加到 `CACHED_URLS` 数组中，以确保在 service worker 安装时会将其缓存下来。

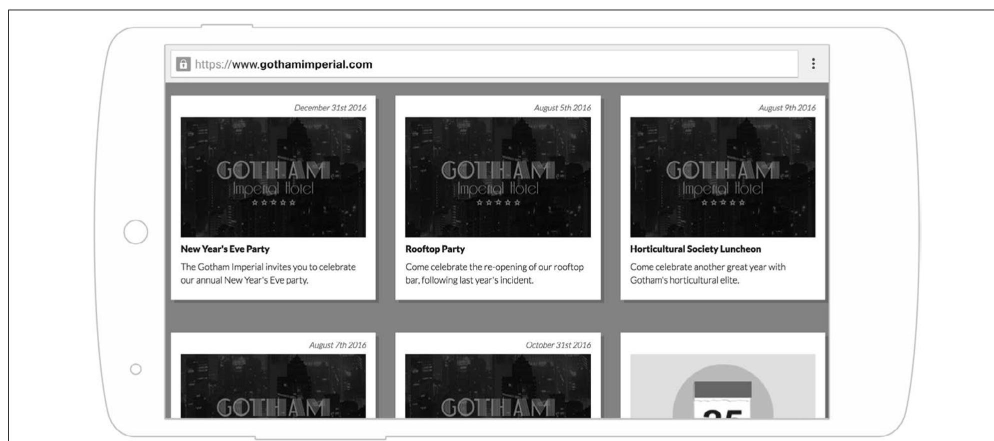


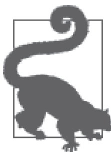
图 5-1：当在缓存中找不到图片时，显示回退图片

我们添加的最后一个条件是处理谷歌分析的请求，让其总是响应网络内容：

```
if (requestURL.host === "www.google-analytics.com") {
  event.respondWith(fetch(event.request));
}
```

这段代码有点多余。我们可以简单地删除它，让浏览器使用默认行为（作用与此处硬编码的内容一致）来处理请求以达到相同的效果。

在哥谭帝国酒店的例子中，可以删除这段代码。但在你的应用中，某些情况下需要明确定义这一点。例如，代码中可能会有一处处理所有请求的 `catch-all`，你需要通过这种方式处理一些异常。



华盛顿邮报的预测缓存

对华盛顿邮报团队而言，让用户在每次访问时阅读更多文章是至关重要的，确保页面加载尽可能快就是关键。

通过运用上述的各种模式，华盛顿邮报团队已经从他们的渐进式 Web 应用中获得许多性能收益，而且，他们还创新性地得到了最大的速度提升——预测缓存。

当你在华盛顿邮报网站阅读文章时，网站的 service worker 选择缓存的不是网站上最流行的文章，也不是你当前阅读的分类中最流行的文章，而是将你最有可能阅读的下一篇文章（也就是直接从当前文章可以直达的链接）中的文本、图片甚至是视频的前几秒缓存下来。

仅凭此改变，团队就让下一篇文章的加载时间缩短到 100 毫秒左右。这一改进直接有助于提升每月文章阅读量和广告浏览量，并最终获得更多的订阅支付用户。

5.6 App shell架构

目前为止，规划缓存策略时，我们提出的方案比较适合于内容站点。但是，许多渐进式 Web 应用看起来并不像传统的内容网站，反而更类似于原生应用。现在，我们将注意力放在如何缓存并提供更为动态的 Web 应用上。

目前为止，我们使用过的工具和技术依然适用于此。我们将采用所学到的一切，实现一个更适用于 Web 应用的缓存策略——App shell 架构（application shell architecture，简称 App shell）。

App shell 架构不是一个革命性的想法。事实上，可能你已经使用了类似的方法构建你的 Web 应用。许多的 JavaScript 框架都已经强制将应用内容、用户界面，以及加载、显示和控制二者所需要的逻辑分离。App shell 架构鼓励你进一步将渲染应用大部分基础界面所需的基础逻辑和资源，与应用的其他部分隔离开来。它还鼓励你尽可能轻量地向用户呈现一个 shell，随着其变得可用再填充内容和附加功能。它会优先显示屏幕上方的结构与内容，而不是那些可以推迟处理的结构。

App shell 架构的目标是尽快向用户提供有意义的体验。一个实现了 App shell 架构且设计良好的渐进式 Web 应用，会在毫秒级内加载并显示其基础界面。



让我们将注意力转回到我们的消息应用中。可以认为，这款应用的最小 shell 就是一个头部，其中包含应用图标、基本元素以及可以输入新消息的输入框（见图 5-2）。

下图左侧显示的最小 shell，可以在用户首次访问时，非常快速地加载。随后会将其缓存起来，以便在用户随后的访问中，都可以在毫秒级内加载。一旦这个 shell 渲染完成，应用就可以从网络加载新的内容以及一些额外的脚本，以开启应用的其余功能。

这种策略可以让你创建一个几乎实时响应的应用。它在第一时间就给用户呈现了界面，而不是让用户盯着一个空白屏幕并等待网络做出响应。用户甚至可以马上开始输入新消息，其他内容和功能将会在后台加载。

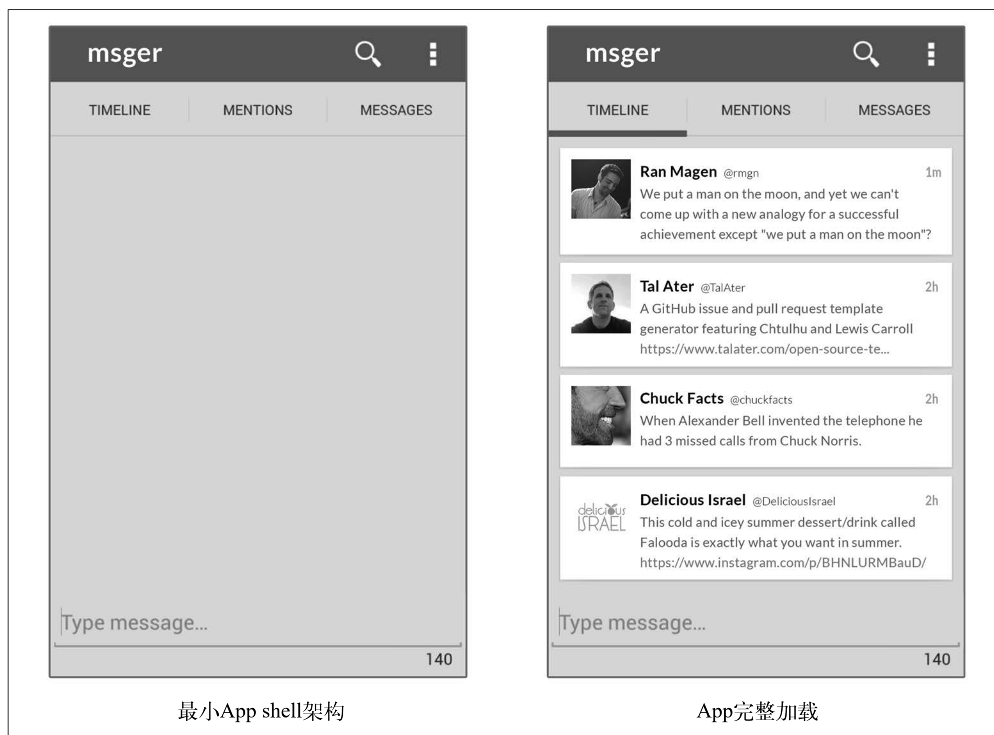


图 5-2：App shell 架构和完整功能的对比

在规划 App shell 架构时，努力为渲染基础用户界面提供尽可能少的 HTML、CSS、JavaScript 和图片，让这个 shell 尽可能地精简，这样当用户第一次访问应用时，它就可以尽快加载并运行。然后，它应该立即存储到缓存中，以便在后续的访问中可以在网络调用之前加载。这样可以使得用户界面在几毫秒内完成加载。一旦初始 shell 呈现给用户后，应用就可以填充内容，并扩展更多功能。

不要忘了，Web 的核心优势之一是能够直接链接到内容。在规划 App shell 架构时，要记住用户可能不会总是从首页开始访问。App shell 应该兼顾用户在首页或用户管理页开始访问。在图 5-2 中可以看到，App shell 无论在时间线、他人提及还是在消息页启动，都是可用的。

拥抱这一策略后，你就可以创建几乎即时加载的应用，在第一时间向用户呈现内容，而不是让用户盯着空白页面，等待网络响应。你创建出的用户体验将会更加接近原生应用，而不是传统的 Web 体验。

在初始渲染时包含内容

没有任何规则声明，使用了 App shell 架构的应用在第一次渲染页面时应该只显示一个空 shell，然后等待网络显示任何内容。不同情况有不同的处理，或许对于你的应用来说，在初始渲染时将缓存内容和 shell 一起渲染会更加合适，即使内容可能是过时的。

在将内容包含到初始渲染之前，先要问自己两个问题。

(1) 渲染缓存中可能过时的内容，然后在几秒钟之后使用网络中的新内容替换它，对于用户体验是一种损害还是一种提升？
(2) 检索并渲染缓存内容，会显著影响应用的初始加载时间和渲染速度吗？



在我们的示例消息应用中，我们可以认为，将缓存中存储的旧消息作为初始渲染的一部分，可以改进用户体验。已经缓存的消息可以很快被渲染，当新消息到达时，将旧的消息推往下方，这也是应用正常流程的一部分（见图 5-3）。

第 6 章会讨论如何将这类消息的数据存储到本地数据库中，并使用它们来填充具有内容的 App shell。

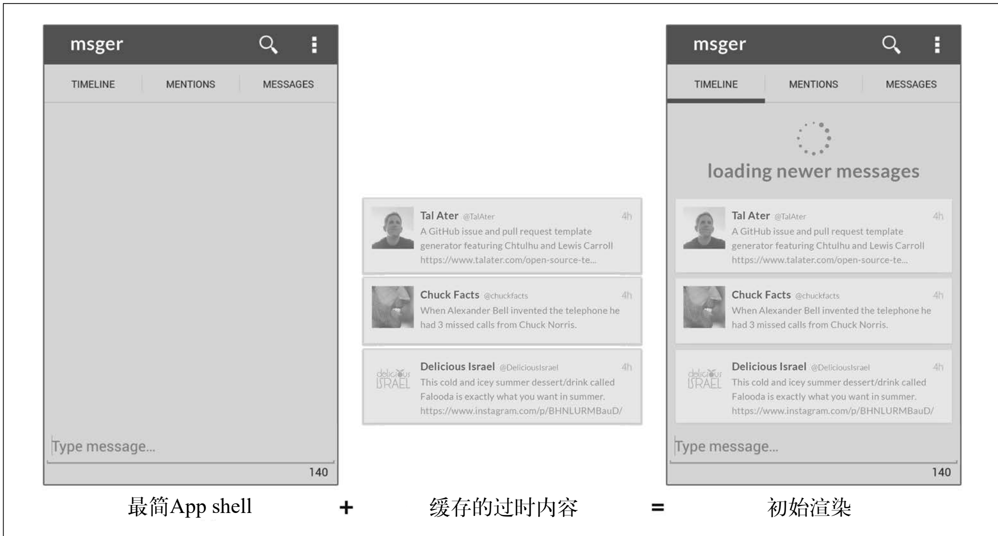


图 5-3：在初始渲染中，将 App shell 和缓存内容相结合

正如我们看到的那样，并没有什么硬性规则告诉我们 App shell 应该或者不应该做什么（也没有一个适合所有应用的架构）。在规划应用的基本 shell 时，可以问问自己：哪些组件对于初始渲染是至关重要的？哪些组件不依赖于数据变化，并且总是可以从缓存中提供的？是否有一些复杂逻辑，可以在基础界面渲染之后才延迟加载？原生应用的开发者是如何应对这些情况的？

5.7　实现App shell

目前为止，我们只关注了哥谭帝国酒店的首页。让我们将注意力转移到用户账号页面。

用户账号页面会在用户点击应用右上角的“My Account”链接，或者用户试图发起新的预订时显示。这是一个简单的单页应用，其中包含了用于预订的控件，并且会加载和渲染一个事件列表以及用户的预订。

这个页面非常适合采用 App shell 架构。

在规划缓存策略时，首先要查看构成应用的不同组件。应用中的哪些部分可以被缓存下来，并且作为 App shell 的一部分立即渲染呢？

(1) 页面的基础布局包含了简单的 HTML 标记和简单的样式表。这两者都可以被缓存，渲染相对较快。
(2) 页眉和页脚包含了一个酒店标志的 PNG 文件。由于这个标志是酒店品牌的重要组成部分，并且文件体积相对较小（7KB），我们将其包含在 App shell 中。
(3) 页眉包含了一张哥谭天际线的大背景图。这是一个可以延迟加载的典型示例，我们不需要将其包含在 App shell 中。
(4) 预订列表和事件列表的数据都是通过 Ajax 进行加载的。这些内容可以在初始 App shell 加载并渲染之后，再添加到页面中。

我们的账号页面已经被构造成：先渲染一个最小 shell，随后动态加载剩余内容（见图 5-4）。现在要实现缓存策略就简单了，只需要在 service worker 安装时，缓存三个额外的请求（账号页面的 HTML、其 JavaScript 文件，以及预订的 JSON 文件），并且在 `fetch` 事件监听器中添加两个条件，处理 my-account.html 和 reservations.json 文件的请求即可。

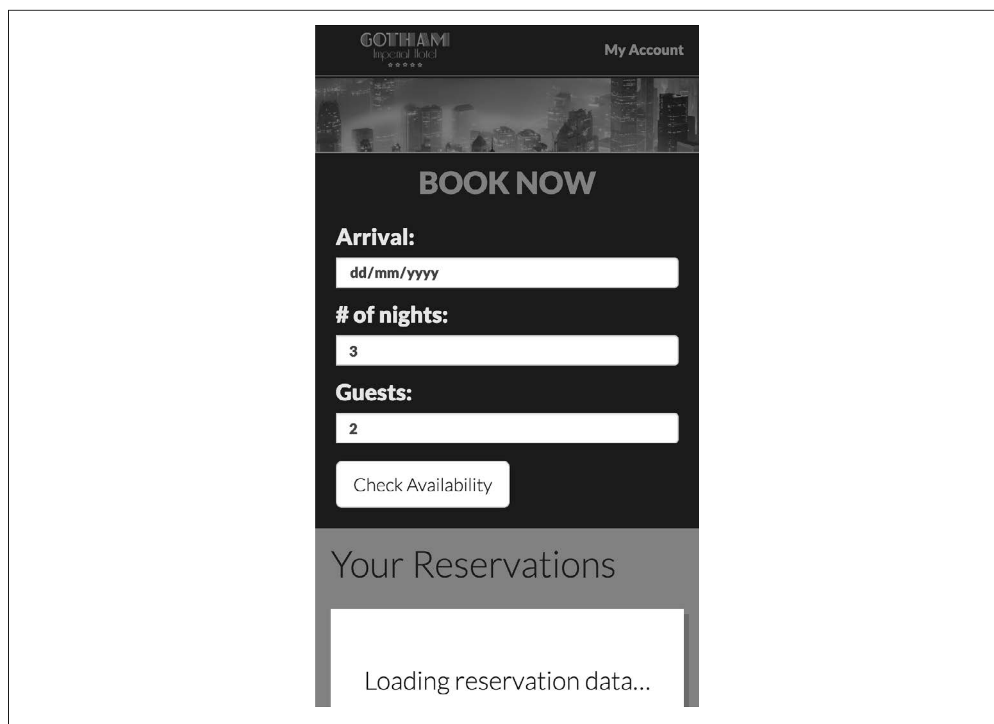


图 5-4：账号页面的空 App shell

在 serviceworker.js 中，添加 my-account.html、/js/my-account.js 和 /reservations.json 到 `CACHED_URLS` 数组中。

同时，在该文件中，在 `fetch` 事件监听器检查 index.html 请求条件的后面，补充下列两个条件：

```
} else if (requestURL.pathname === "/my-account") {
  event.respondWith(
    caches.match("/my-account.html").then(function(response) {
      return response || fetch("/my-account.html");
    })
  );
} else if (requestURL.pathname === "/reservations.json") {
  event.respondWith(
    caches.open(CACHE_NAME).then(function(cache) {
      return fetch(event.request).then(function(networkResponse) {
        cache.put(event.request, networkResponse.clone());
        return networkResponse;
      }).catch(function() {
        return caches.match(event.request);
      });
    })
  );
```

在第一个条件中，使用**缓存优先，网络作为回退方案**模式返回了 My Account 页面的 HTML 内容。第二个条件通过**网络优先，缓存作为回退方案，并频繁更新缓存**模式提供 reservations.json 文件（类似于缓存并提供 events.json 的方式）。

通过这些微小的修改，现在 My Account 页面的 App shell 可以在毫秒级内向用户提供有意义的体验。酒店的所有品牌元素以及屏幕上方的大部分内容都将立即呈现。预订的小部件立即可用，并且可以毫无延迟地使用。初始 shell 渲染完成后，动态的预订和事件数据才会被加载和显示。

5.8 解锁成就

是的，能够将我们的应用渲染给离线用户非常棒。但除此之外，本章还极大地改善了有连接用户的体验。

抛开对离线用户的明显改进，很容易忽略我们在本章中取得的所有成就，以为只是一点小改进。毕竟，我们正在通过一种不现实的快速连接本地服务器的方式来查看网站。

让我们来获取一些新的想法。

在 Chrome 开发者工具中，我们可以模拟不同的连接速度（参见 4.8 节了解详情），并精确地测量结果。

让我们来看看使用 3G 连接访问首页的加载情况。

	DOM ready事件耗时	总加载时间	带宽使用
不使用 SW，首次访问	1200 毫秒	13 秒	1.1MB
不使用 SW，二次访问	587 毫秒	4.9 秒	357KB
使用 SW，首次访问	1200 毫秒	13 秒	1.1MB
使用 SW，二次访问	155 毫秒	1.1 秒	29.7KB

不管是否使用 service worker，用户首次访问页面都需要经过 13 秒的加载时间，带宽的使用量也接近。但是一旦用户浏览过我们的网站，后续每次访问的差异就会使人大吃一惊。

- 使用 service worker 后，页面加载速度提升至原来的 4.5 倍。
- 使用 service worker 后，DOM ready 事件的触发速度提升至原来的 3.8 倍。
- 用户需要下载的数据量减少了 91%，大大降低了用户的成本以及我们的托管成本。

用户从首页导航到账号页面也能看到巨大的收益。

	DOM ready事件耗时	总加载时间	带宽使用
不使用 SW	578 毫秒	1 秒	28.9KB
使用 SW	150 毫秒	0.6 秒	14.9KB

DOM 完全加载的时间缩短了 428 毫秒，看起来无足轻重，但是在我看来绝非如此。用户点击“My Account”按钮时，下一屏幕的加载时间（从 578 毫秒到 150 毫秒）的差异，就相当于 Web 页面跳转和原生页面跳转的时间差异。在第 6 章中，我们将进一步改进这个页面。

只需几个基础的构建模块以及一些常见的缓存模式，我们就能用短短几行代码取得非凡的效果，既提高了用户体验，又减少了带宽的使用量（为用户省钱的同时，也降低了我们的成本）。



要创建离线优先的应用，除了处理连接的变化，还需要做更多工作。

当我们改进了离线功能的支持时，也出现了新的用户体验挑战。例如：

- 我们如何告知线用户，他看到的是缓存内容而且可能是过时的。
- 我们如何保证用户的修改即使在断开连接的情况下也不会丢失？

在第 11 章中，我们将会更深入地探索这些 UX 挑战。

5.9 小结

本章给予我们一些很好的机会来学习常见的缓存模式，并在“离线优先”的路上逐一运用这些模式来解决不同的挑战。

希望本章所概述的不同缓存模式可以帮助你一路前行。请记住，这些模式不是固定的模板。有时候你需要组合并配对使用（例如，我们的事件图片代码既使用了按需缓存，又实现了通用回退模式），有时候你还需要提出一些完全不同的方案（例如，如果客户端不支持通用回退图片，我们可能要考虑在安装阶段解析 events.json 文件，并缓存其中的图片）。

要解锁离线优先，是没有任何预设的公式适合你的 Web 应用的。在规划应用策略时，始终要考虑每个资源是如何使用、何时使用的。要权衡每个资源保持最新对于性能的潜在收益与影响。

始终首先考虑用户的行为与需求，并以此为指导，找出最佳用户体验的实现。

第6章

使用IndexedDB在本地存储数据

目前为止，我们使用 CacheStorage 存储所有文件以及应用所需的数据。当想要存储一份预订列表时，我们简单地在 CacheStorage 中缓存了一个 HTTP 响应，其中包含了一个 JSON 数据文件。随后我们可以在缓存中取出该文件，并且在每次访问预订列表时，解析这份数据。但是，如果用户发起了新的预订，或者预订状态发生了改变，会发生什么呢？这种情况下，我们不得不求助于服务器，加载一个最新的 JSON 文件。

虽然我们的应用目前在任何连接状态下都可以进行操作，但是，如果连最简单的数据操作都依赖于服务器，是否真的可以称之为离线优先呢？

我们需要一种更好的方法来在浏览器中持久地处理数据。这种方法应该可以让我们在本地存储、读取和修改数据，而不依赖于网络状态。

在本章中，我们将把这个重要的工具添加到工具集，学习如何使用一个称为 IndexedDB 的本地数据库。

和服务端数据库一样，IndexedDB 允许我们以结构化的方式存储、查询并修改数据等。和服务端数据库不同的是，IndexedDB 可以完全在浏览器中实现这一切操作。

本章首先会概述 IndexedDB 及其语法，我们在此试验的代码将与哥谭帝国酒店无关。随后，我们会结合所学的知识，在哥谭帝国酒店应用中实现它们。

在本章结尾，我们将会添加一个本地数据库到哥谭帝国酒店应用中，无论用户连接状态如何，它都可以正常工作。我们将会拥有一个可以在毫秒级别内加载的应用，并在不依赖服务器的情况下显示内容并操作数据。和原生应用类似，我们的渐进式 Web 应用仅在从服务器获取更新的数据和内容，或者将用户操作传回给服务器时，才需要连接到网络。

6.1 什么是IndexedDB

IndexedDB 是浏览器中的事务型对象存储数据库。

在这个关键词丰富、定义令人困惑的描述（它可以轻易地容纳很多标签，让整个营销部门忙碌数周）背后，其实是一些简单的概念。让我们将这句话拆解开来，逐一解释各个词的含义。

IndexedDB是事务型的

你在 IndexedDB 中执行的操作会按照事务来分组。在一个事务中，要么所有操作都成功，要么所有操作都失败。

假设你的数据库是为银行网站的用户存储余额的。如果你试图进行一项“交易”，将 7 美元从 Jill 转账给 Jake，这笔交易会包含两个操作。

(1) 从 Jill 的账户减去 7 美元。
(2) 给 Jake 的账户增加 7 美元。

如果 Jill 只有 2 美元，第一个操作就会失败，但第二个操作成功，这样你只会为银行创造了 7 美元的赤字。如果第一个操作成功，但第二个操作因为 Jake 的账户被冻结而失败，你就抹去了 7 美元的存在。通过把操作组合成单个事务，我们就可以确保：要么所有操作失败，钱不会被转移；要么所有操作成功，银行可以收取 6 美元的手续费。

IndexedDB是对象存储数据库

与传统的关系型数据库（例如 MySQL 和 SQL Server）包含了预定义数据列、按行存储的表格不同，对象存储数据库是存储对象的。每个数据库可以包含多个对象存储，每个对象存储可以包含多个对象。这些“对象”可以是 JavaScript 对象、布尔值、数字、二进制块，以及 JavaScript 可以处理的大部分其他数据单元。

你可能已经熟悉另一个常用于描述这类数据库的词——NoSQL。

在上述的银行数据库例子中，可能包含了一个客户对象的存储，其中包含了很多对象，每个对象代表一个客户。每个客户对象包含姓氏、名字、余额、最后登录时间以及最后 10 次的存款记录。

IndexedDB是索引数据库

和传统的关系型数据库系统类似，IndexedDB 也使用了索引。你可以在任何对象存储中添加索引，并使用它来检索需要的对象。

在我们的示例银行客户对象存储中，可能包含了一个用户姓氏的索引。这样我们就可以轻松找到姓氏为 Dwayne 的用户。它还可能包括了最后登录时间的索引，让我们可以检索到最后 10 个登录到应用的用户。

IndexedDB是基于浏览器的

IndexedDB 完全基于浏览器运行。不管用户连接状况如何，都可以访问并操作已经存储其中的任何数据。

这是一把双刃剑。你在本地数据库中的任何修改都不会自动反映到服务器。如何将本地修改上传到服务器，或者从服务器同步变更到本地数据库，完全取决于你。

有很多开源库可以简化本地数据库和服务器之间的数据传播。在 6.8 节中，我们会探索其中几个。

除了上述这些 IndexedDB 核心概念之外，在使用 IndexedDB 时，还需要牢记更多的内容。

- 你可以创建多个数据库（虽然大部分应用通常只会创建一个数据库）。
- 每个数据库中可以包含多个对象存储。
- 每个对象存储通常只包含一种数据类型（例如用户类型、聊天消息类型、预约类型等）。
- 对象存储包含键值对。
- 值几乎可以是任何可以用 JavaScript 表示的内容，包括对象、数字、布尔值、字符串、日期、数组、正则表达式、undefined 和 null。
- 键用来引用对象存储中的某个值。键可以是简单的数字标识符，或者是指向值中的特定路径。例如，如果我们存储用户数据，每个值是一个包含了姓氏、名字、护照编号的对象，那么我们可以把护照编号作为对象的键。
- IndexedDB 遵循**同源策略**，确保用户在访问任意 Web 站点时，不必担心数据会被另一个网站读取。换句话说，你可以在你的域中读写数据，但是不能访问或者写入另一个域的 IndexedDB 数据。
- 数据库是版本控制的。如果你想要创建对象存储，或者修改其结构，可以通过新的版本号打开数据库连接。这会触发一个 `upgrade needed` 事件。在这个事件期间，可以进行此版本和旧版本之间的任何数据库更改。
- 大部分的 IndexedDB 操作都是异步的。如果你请求获取一个值，API 不会简单地返回这个值。反之，你需要定义一个回调函数来处理这个事件。当回调函数被调用时，它将会包含你所请求的值。

如果你以前使用过 NoSQL 数据库，可能对其中大多数概念已经很熟悉了。即使你没有，使用 IndexedDB 也是很简单的。

大部分和 IndexedDB 的交互都可以提炼成以下这种基本模式，你将会反复使用到。

(1) 打开数据库。
(2) 启动事务，用来读取或写入对象存储。
(3) 打开对象存储。
(4) 在对象存储中执行操作（检索对象、添加对象、删除对象等）。
(5) 完成事务。



IndexedDB 浏览器支持

从历史上看，IndexedDB 声名狼藉。这主要是因为在 Safari 以及 iOS 8 和 iOS 9 上其实现非常糟糕，并且在 iOS webview 中完全缺乏支持。

幸运的是，IndexedDB 最近的遭遇已经好多了。截至 2017 年本书（英文版）出版时，IndexedDB 可以在大多数现代浏览器中运行良好（除了 Opera Mini）。

对于需要支持旧浏览器的场景（包括 IE 9 或以下、安卓浏览器 4.3 或以下），6.8 节探索了一系列的库，它们可以通过回退到替代技术的方式支持旧浏览器，例如 WebSQL 和 localStorage。

即使你选择忽略旧浏览器，也最好在使用 IndexedDB 之前进行功能检测。在 6.4 节中，可以看到这一点。

6.2 使用IndexedDB

虽然 IndexedDB 因为有点混乱而声名狼藉，但是我们会采取一种实用的、动手实践的方式来快速理解它背后的核心原理，并在短时间内取得具体的结果。

在 6.8 节中我们会看到一些实用的库，它们令使用 IndexedDB 的体验更加愉快。

让我们现在开始探索吧。



IndexedDB

本节中的代码用于一般性地阐述 IndexedDB，而不属于哥谭帝国酒店网站。

要跟随代码，可以在你喜欢的代码编辑器中打开 /public/indexeddb.html，并在 `<script>` 标签中进行修改。下一步，在开发服务器已经运行的前提下（在 2.4 节中已做解释），在浏览器中打开 http://localhost:8443/indexeddb.html 即可。

我们将在 6.4 节中回到哥谭帝国酒店。

6.2.1 打开数据库连接

使用 IndexedDB 的第一步是打开数据库连接。

在 indexeddb.html 中添加下列代码：

```
var request = window.indexedDB.open("my-database", 1);

request.onerror = function(event) {
  console.log("Database error: ", event.target.error);
};

request.onsuccess = function(event) {
  var db = event.target.result;
  console.log("Database: ", db);
  console.log("Object store names: ", db.objectStoreNames);
};
```

即使是最基本的 IndexedDB 代码示例，也体现了 IndexedDB 的异步性质。调用 `window.indexedDB.open()` 之后，并没有返回一个数据库连接。取而代之的是，它返回一个打开数据库连接的请求。随后我们可以监听这个请求的事件，例如 `success` 或者 `error` 事件。

当你在浏览器中运行这段代码时，会在浏览器中创建一个名为 `my-database` 的数据库，并

打开（如果已经存在，则直接打开，不会执行创建）。随后会触发 success 事件，我们可以在其中使用 IDBDatabase 对象，将其打印到控制台，并且连同这个全新数据库中的对象存储列表一起打印出来。

由于我们的数据库现在仍然是空的，所以这对于我们没有作用。让我们添加一个对象存储到数据库中，其中包含一份银行客户列表。

6.2.2 数据库版本/修改对象存储

和 service worker 非常相似，IndexedDB 数据库也是版本化的。每当我们想要修改数据库的结构，例如添加、修改或删除对象存储时，就需要创建一个新版本。

我们可以通过增加版本号并作为第二个参数传递给 indexedDB.open()，来创建一个新的数据库版本。当浏览器检测到版本号大于现有版本时，就会触发 upgrade needed 事件。我们可以监听这个事件，并使用它来修改数据库。

在上述示例代码的末尾，添加下列代码：

```
request.onupgradeneeded = function(event) {
  var db = event.target.result;
  db.createObjectStore("customers",
    { keyPath: "passport_number" }
  );
};
```

当数据库触发 upgrade needed 事件时，这段代码就会执行。它可以从事件中获取数据库对象，并在其中创建一个名为 customers 的新对象存储。它还使用了 keyPath 属性，定义了护照编号作为存储中每个对象的唯一键。

刷新页面，并查看记录到控制台中的数据库对象。

```
Database:  ▶IDBDatabase {name: "my-database", version: 1, objectStoreNames: DOMString…}
Object store names:  ▶DOMStringList {length: 0}
>
```

控制台中的第二行清晰地显示出，我们的数据库中仍然没有包含任何对象存储。这是为什么呢？答案在第一行。数据库依然是版本 1，所以我们的 upgrade needed 事件没有触发。

让我们将代码中第一行的版本号修改为 2：

```
var request = window.indexedDB.open("my-database", 2);
```

刷新页面并查看控制台，现在应该可以看到，数据库已经成功更新到版本 2，其中包含了一个名为 customers 的对象存储：

```
Database:  ▶IDBDatabase {name: "my-database", version: 2, objectStoreNames: DOMString…}
Object store names:  ▶DOMStringList {0: "customers", length: 1}
>
```

6.2.3 添加数据到对象存储

要让数据存储发挥作用，还需要让其存储一些对象。让我们往里添加几个用户。

在浏览器中打开 http://localhost:8443/indexeddb.html 之后，在浏览器控制台中运行以下代码：

```
var request = window.indexedDB.open("my-database", 2);

request.onsuccess = function(event) {
  var db = event.target.result;
  var customerData = [
    {"passport_number": "6651", "first_name": "Tal", "last_name": "Ater"},
    {"passport_number": "7727", "first_name": "Archie", "last_name": "Stevens"}
  ];
  var customerTransaction = db.transaction("customers", "readwrite");
  customerTransaction.onerror = function(event) {
    console.log("Error: ", event.target.error);
  };
  var customerStore = customerTransaction.objectStore("customers");
  for (var i = 0; i < customerData.length; i++) {
    customerStore.add(customerData[i]);
  }
};
```

这段代码创建了一个新的 `readwrite` 读写事务，并将其作用域设置为 `customers` 对象存储。它还监听了事务的错误事件，并将其打印到控制台中。随后它使用了事务的 `objectStore()` 方法，打开 `customers` 对象存储，并继续使用该对象存储的 `add()` 方法往里添加两条记录。



启动事务

如前所述，在 IndexedDB 中进行的大部分操作都是事务型的。在添加数据到对象存储之前，我们需要启动一个新的事务。

事务可以通过在数据库对象上调用 `transaction()` 方法启动，事务的作用域作为传入的第一个参数。`transaction()` 方法还接受一个可选的第二参数，这个参数可以控制事务是 `readonly`（只读事务，默认值）还是 `readwrite`（读写事务）。如果要在该事务期间添加、删除或者修改对象存储中的数据，则需要打开一个 `readwrite` 事务。

事务的作用域可以是字符串，或者包含对象存储的字符串数组。IndexedDB 通过定义事务的作用域，避免了不同事务之间的竞争条件（例如，两个或多个事务试图在同一时间修改同一对象存储）。如果创建了两个或多个具有重叠作用域的 `readwrite` 事务，它们将会进入队列，串行运行。如果它们的作用域不同，或者是 `readonly` 事务，则它们可以并行运行。

在浏览器控制台中运行一次上述示例代码之后，我们将尝试再次运行，但在此之前，我们先要做出一点修改。修改上述示例中的代码，将第二个用户（Archie）的 `passport_number` 属性修改成一个不同的值。现在尝试再次在控制台中运行新的代码。

你应该会看到两条错误消息：

```
Error:  DOMException: Key already exists in the object store.
Error:  DOMException: The transaction was aborted, so the request cannot be fulfilled.
>
```

这两处错误阐明了 IndexedDB 的两个核心概念。

第一处错误抛出是因为我们设置了 `customers` 对象存储是使用 `passport_number` 值作为键的。这意味着 `passport_number` 值必须是唯一的。当我们尝试添加一条与现有记录 ID 相同的新记录时，IndexedDB 就会抛出这个错误。

第二处错误清楚地表明了 IndexedDB 的事务性质。尽管第二个对象的唯一 ID 是有效的，但是它依然没有添加到数据库中，因为先前的操作失败了。事务保证了要么所有操作成功，要么不进行任何操作。

6.2.4 从对象存储中读取数据

现在我们在对象存储中拥有了前两个客户，让我们来学习如何从中检索对象。

读取数据有三种方法。你可以使用键名来检索单个对象，可以使用游标来遍历存储中的所有对象，也可以使用索引来检索数据的较小子集（然后使用游标来遍历）。

我们首先使用键名来从对象存储中读取单个对象。

在浏览器控制台中运行下列代码：

```
var request = window.indexedDB.open("my-database", 2);

request.onsuccess = function(event) {
  var db = event.target.result;
  var customerTransaction = db.transaction("customers");
  var customerStore = customerTransaction.objectStore("customers");
  var request = customerStore.get("7727");
  request.onsuccess = function(event) {
    var customer = event.target.result;
    console.log("First name: ", customer.first_name);
    console.log("Last name: ", customer.last_name);
  };
};
```

假设你运行过前面列出的代码，往 `customers` 对象存储中添加了几个客户，这段代码应该能够通过匹配护照编号，检索出特定的用户，并将其姓名打印到控制台中。

和大部分的 IndexedDB 操作类似，我们首先要打开数据库，并创建一个新的事务。和以前一样，我们将事务的作用域限制在 `customers` 对象存储，但这一次我们不需要传递 `readwrite` 标记了。因为我们无意在事务中写入任何内容，所以使用 `readonly` 事务就足够了。

随后我们在对象存储上调用了 `get()` 方法，传入一个键名（护照编号），这个键名会和想要查找的客户对象相匹配。由于 `get()` 是一个异步操作，它不会马上返回结果，而是返回一个代表了请求的对象。通过监听这个请求的 `onsuccess` 事件，我们就可以等待请求完成，并返回所请求的对象。



你可以互相串联大部分的 IndexedDB 方法，以创建出更短、更简洁的代码。如果你在随后不需要引用 transaction、objectStore、get 等方法创建的特定对象，这种解决方案会非常不错。

通过互相串联方法，上述示例代码中的 request.onsuccess 可以精简为：

```
request.onsuccess = function(event) {
  event.target.result
    .transaction("customers")
    .objectStore("customers")
    .get("7727")
    .onsuccess = function(event) {
      var customer = event.target.result;
      console.log("First name: ", customer.first_name);
      console.log("Last name: ", customer.last_name);
    };
};
```

6.2.5 IndexedDB版本管理

目前为止，我们的数据库只有两个版本。初始版本是空的，没有包含对象存储，而第二个版本添加了一个对象存储。

如果你将数据库版本升级为 3，并刷新页面，会发生什么呢？你将会收到以下错误提示：

```
Failed to execute 'createObjectStore' on 'IDBDatabase':
An object store with the specified name already exists.
```

让我们来试图了解发生了什么。当你第一次加载页面时，代码会创建数据库并提供版本号 1，随后尝试运行 onupgradeneeded 方法。这时候，我们还没有定义这个方法，创建的数据库是空的。随后，我们添加了 onupgradeneeded 方法，这个方法创建名为 customers 的数据存储，并将版本号修改为 2。当我们刷新页面时，数据库注意到版本号大于当前版本，并运行 onupgradeneeded 方法，创建名为 customers 的数据存储。最后，我们将版本号更新为 3。当我们刷新页面时，数据库再一次注意到版本号变化，并再一次运行 onupgradeneeded 方法。然而，这次试图创建的对象存储已经存在，就会导致错误。我们的数据库会停留在版本 2，因为 onupgradeneeded 事件失败了。

不幸的是，由于版本 3 依赖这个数据存储，我们不能简单地从 onupgradeneeded 方法中删除这段代码。如果我们这样做了，那些自从版本 1 后没有访问网站的用户（或者是首次访问的用户）就不会拥有这个对象存储。我们需要一种依赖当前状态，有条件地改变数据库的方法。

解决这个挑战的一种方式来源于传统的数据库世界——**迁移**（migration）。迁移由一系列的原子步骤组成，每个步骤都可以将数据库向前移动一个版本。

下面是实现 IndexedDB 迁移的一种可能方案：

```
request.onupgradeneeded = function(event) {
  var db = event.target.result;
  var oldVersion = event.oldVersion;
  if (oldVersion < 2) {
```

```
      db.createObjectStore("customers",
        { keyPath: "passport_number" }
      );
    }
    if (oldVersion < 3) {
      db.createObjectStore("employees",
        { keyPath: "employee_id" }
      );
    }
  };
```

通过检查数据库先前的版本号，这个方法可以将数据库从任意版本逐步带到最新的版本。如果用户首次访问站点（`oldVersion == 0`），两个迁移都会运行。如果用户没有访问过网站的上一个版本（`oldVersion == 2`），只有第二个迁移会运行。

虽然这种方式确实可以将数据库从版本 1 逐步精确地重新创建回到最新版本，但是如果要维护很多个版本，这种方法很快就会失控了。

在版本之间升级数据库的另一种方法，是测试当前数据库的状态，并且根据需要进行修改。

在 indexeddb.html 中，将 `onupgradeneeded` 方法更新为下列代码，并确保脚本的第一行被设置为打开数据库版本 3：

```
  request.onupgradeneeded = function(event) {
    var db = event.target.result;
    if (!db.objectStoreNames.contains("customers")) {
      db.createObjectStore("customers",
        { keyPath: "passport_number" }
      );
    }
  };
```

刷新浏览器，现在你的数据库应该能够升级到版本 3，并且不会抛出错误。

通过这种方式，在进行修改之前，你总是需要检查是否需要进行修改。只有数据存储不存在的时候，才会去添加它。只有当索引已经存在时，才会去删除它。

管理 IndexedDB 版本没有唯一正确的方法。你可能会发现，在不同项目中，这两种方法可能各有千秋。你甚至还会组合使用这两种方法（例如使用第二种方法来升级数据库结构，然后使用迁移将所有客户对象的名字大写，如果版本号在 19 之前）。

6.2.6　使用游标读取对象

我们已经看到了如何使用 `get()` 方法从对象存储中检索单个对象。不幸的是，这种方法只有当检索单个对象，并且知道其确切的键名时，才能奏效。要检索多个对象，需要打开游标。



游标是什么？

如果你熟悉基于 SQL 的数据库，可以将打开游标视为运行一个 `SELECT * FROM table` 查询。就像这个查询可以通过 `WHERE` 和 `LIMIT` 参数加以修饰那样，游标也可以通过指定索引和边界来修改。

和 SQL 返回的结果不同，游标不会包含结果。它只是指向对象存储中实际对象的指针列表。在任何时候，游标只会指向对象存储中的一条记录，当你告诉它进行 continue() 或者 advance() 操作时，可以将其向后或者向前移动。这使得我们可以在大型对象存储中进行迭代（或者遍历），而不需要将所有对象存储在内存中（见图 6-1）。

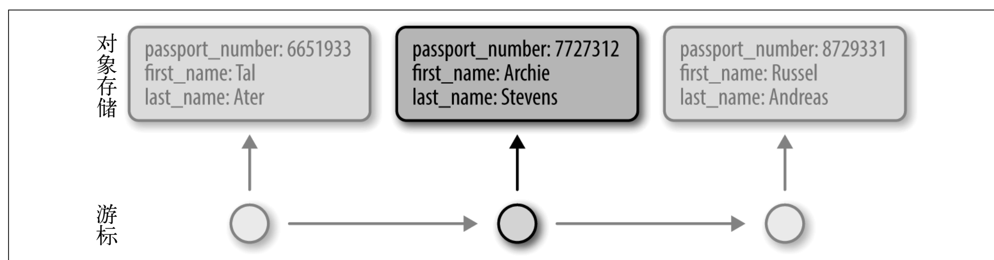


图 6-1：指向对象（但不包含对象）的游标

让我们来打开第一个游标。在浏览器控制台中运行下列代码：

```
var request = window.indexedDB.open("my-database", 3);

request.onsuccess = function(event) {
  var db = event.target.result;
  var customerTransaction = db.transaction("customers");
  var customerStore = customerTransaction.objectStore("customers");
  var customerCursor = customerStore.openCursor();
  customerCursor.onsuccess = function(event) {
    var cursor = event.target.result;
    if (!cursor) { return; }
    console.log(cursor.value.first_name);
    cursor.continue();
  };
};
```

假设你已经运行过前面示例中列出的代码，添加了几条客户数据到 customers 数据存储中，那么这段代码会迭代所有客户，并将客户的名字打印到控制台。

前面几行代码对你来说应该很熟悉了。我们打开数据库，启动一个新的事务，并打开了 customers 数据存储。

接下来，我们继续在对象存储上调用了 openCursor() 方法，这个异步方法会打开一个新的游标，并且在每次游标前进时触发 onsuccess 事件。

在 onsuccess 函数中，我们可以访问游标（通过 event.target.result 可以找到）来检索当前指向的对象。我们把对象的 first_name 属性值打印出来，并且让游标调用 continue() 指向下一个对象。每当游标改变时，会触发另一个 onsuccess 事件，它将再次运行我们的函数，并打印下一个客户的名字，以此类推。

需要重点记忆的是，每次游标前进都会触发 onsuccess 事件，甚至在穿过最后一条记录或者对象存储为空时也是如此。此时，游标（event.target.result）会指向 null。因此，在

onsuccess 方法试图访问对象之前，需要先检查游标是否指向对象。在上述例子中，我们使用了条件 if (!cursor) { return; } 来实现这一点。

6.2.7　创建索引

目前为止，我们只学习了如何在对象存储中打开游标并遍历每个对象。如果你只想检索符合某个标准的对象，那么迭代整个对象存储就不是非常有效或者方便了。通过使用索引，我们可以对数据存储进行“查询”，打开一个只会迭代匹配该查询的对象的游标。

要实现这一点，我们先来创建第二个对象存储，用来保存不同货币间的汇率。我们在其中存储的每个汇率对象看起来是这样的：

```
{"exchange_from": "CAD", "exchange_to": "USD", "rate": 0.77}
```

现在，我们想要检索某种货币的所有汇率。解决这个问题的其中一种方法，是检索对象存储中的每个对象并逐一迭代，检查每个对象是否匹配我们想要寻找的货币。而一种更好、更快的方法是使用索引。

在 indexeddb.html 中，将数据库版本修改成 4，并将 onupgradeneeded 方法替换成下列代码：

```
request.onupgradeneeded = function(event) {
  var db = event.target.result;
  if (!db.objectStoreNames.contains("customers")) {
    db.createObjectStore("customers",
      { keyPath: "passport_number" }
    );
  }
  if (!db.objectStoreNames.contains("exchange_rates")) {
    var exchangeStore = db.createObjectStore("exchange_rates",
      { autoIncrement: true }
    );
    exchangeStore.createIndex("from_idx", "exchange_from", { unique: false });
    exchangeStore.createIndex("to_idx", "exchange_to", {unique: false});

    exchangeStore.transaction.oncomplete = function(event) {
      var exchangeRates = [
        {"exchange_from": "CAD", "exchange_to": "USD", "rate": 0.77},
        {"exchange_from": "JPY", "exchange_to": "USD", "rate": 0.009},
        {"exchange_from": "USD", "exchange_to": "CAD", "rate": 1.29},
        {"exchange_from": "CAD", "exchange_to": "JPY", "rate": 81.60},
      ];
      var exchangeStore = db
        .transaction("exchange_rates", "readwrite")
        .objectStore("exchange_rates");
      for (var i = 0; i < exchangeRates.length; i++) {
        exchangeStore.add(exchangeRates[i]);
      }
    };
  }
};
```

我们新的 onupgradeneeded 方法首先确保不会再为已经拥有 customers 对象存储的用户重新创建。同理，我们接下来也会检测 exchange_rates 是否存在，并且在不存在时才会去创建它。



inline key 与 out-of-line key

我们使用自增键创建了 exchange_rates 存储。和 customers 对象存储相比，exchange_rates 没有像护照编号这样的自然唯一标识符。通过设置 autoIncrement 为 true，我们就可以告诉 IndexedDB，让其为每一个对象创建唯一的索引。第一个存储的对象会得到 ID 1，第二个会得到 ID 2，以此类推。

这种键通常称为 out-of-line key，因为键和值的存储是分离的。

使用 keyPath 指向对象自身属性的键称为 inline key。customers 对象的存储就使用了 inline key。

我们的代码在新的对象存储中创建了两个索引。这些索引允许我们打开特定的游标，只迭代符合某些标准的对象。例如，通过使用 from_idx 索引，我们可以检索所有从美元到其他货币的汇率。

createIndex() 方法接收一个索引名作为第一个参数，后面是这个索引需要使用的键（例如 exchange_to），以及一个可选的选项数组。在我们的例子中，我们使用了选项数组，来指定我们用于这个索引的键不是唯一的（即每种货币可以用不同的汇率转换成其他货币）。

在 onupgradeneeded 方法的结尾，我们给 exchange_rates 对象存储填入了一些初始数据。这样做是为了确保一旦对象存储被添加它就是可用的，并且以后不会再次执行这个添加操作（例如从服务器请求更多的最新汇率之后）。请注意，我们要确保只有 db.createObjectStore() 返回的事务完成后，才去添加数据。这样，就可以确保在尝试向对象添加数据之前，成功创建对象存储。

6.2.8 使用索引读取数据

索引允许我们打开特定的游标，只对符合某个标准的结果进行迭代。

在控制台运行下列代码，可以打印出加拿大元兑换所有其他货币的汇率：

```
var request = window.indexedDB.open("my-database", 4);

request.onsuccess = function(event) {
  var db = event.target.result;
  var exchangeTransaction = db.transaction("exchange_rates");
  var exchangeStore = exchangeTransaction.objectStore("exchange_rates");
  var exchangeIndex = exchangeStore.index("from_idx");
  var exchangeCursor = exchangeIndex.openCursor("CAD");
  exchangeCursor.onsuccess = function(event) {
    var cursor = event.target.result;
    if (!cursor) { return; }
    var rate = cursor.value;
    console.log(rate.exchange_from+" to "+rate.exchange_to+": "+rate.rate);
    cursor.continue();
  };
};
```

在打开数据库之后，代码启动事务，并获取了 exchange_rates 对象存储。之前，我们通过打开对象存储自身的游标来迭代整个对象存储。这一次不同，我们首先要从对象存储中获

取索引，随后在索引对象上打开游标。我们通过在对象存储上调用 index() 方法，并传入想要使用的索引名称来实现这一点。随后，我们可以在索引上调用 openCursor() 方法，传入想要寻找的值（在这个例子中，我们想要通过货币的名字获取对应的汇率）。

随后，我们可以通过监听 success 事件来迭代游标，这和遍历对象存储的游标是类似的。唯一的区别是，前者只会迭代符合给定标准的对象（例如，exchange_from 的值为 CAD）。

6.2.9 限制游标的范围

默认情况下，游标会迭代对象存储中的所有对象，或者索引返回的所有对象。你可以通过传递一个 IDBKeyRange 对象，进一步限制游标迭代的对象范围。

上例中的 openCursor 命令可以重写为显式使用 IDBKeyRange。下面的示例中展示了两种方法，它们返回的结果完全一致。传递 IDBKeyRange.only("CAD") 给游标，相当于让其只返回匹配 CAD 索引值的对象：

```
exchangeIndex.openCursor("CAD");
exchangeIndex.openCursor(IDBKeyRange.only("CAD"));
```

除了 only 之外，IDBKeyRange 还支持 lowerBound()、upperBound() 和 bound() 方法。这些方法允许我们将结果限制在一定范围内。

和 only() 方法类似，lowerBound() 和 upperBound() 接收一个值作为第一个参数。这个值会作为范围的下界或者上界。此外，它们还可以接收一个布尔值作为第二个参数，这个参数决定了结果应该排除（true）还是包含（false）那些等于范围边界值的对象：

```
// 包含了所有CAD之上的键，包括CAD本身
// 例如：CAD、USD
IDBKeyRange.lowerBound("CAD", false);
// 包含了所有CAD之下的键，不包括CAD本身
// 例如：AUD、BRL
IDBKeyRange.upperBound("CAD", true);
```

可以将 lowerBound() 和 upperBound() 组合成一条单独的命令 bound()，bound() 同时接收下界和上界作为第一个和第二个参数，布尔值作为第三个和第四个参数，分别对应是否排除那些等于下界或者上界的结果。

下面的代码将返回以字母 C 开头的所有记录的游标（即 C 和 D 之间，包括 C 本身，但不包括以 D 开始的记录）：

```
exchangeIndex.openCursor(
  IDBKeyRange.bound("C", "D", false, true);
);
```

你可以使用 IDBKeyRange 来限制索引或者对象存储上打开的游标返回的结果数。下面这个在对象存储上直接打开的游标，会返回所有符合键大于等于 3 的记录：

```
exchangeStore.openCursor(
  IDBKeyRange.lowerBound(3, false);
);
```

6.2.10 设置游标方向

默认情况下，游标将按照键（对象存储的主键，或者索引键）的升序方向迭代对象。你可以通过在打开游标时传递 prev 作为第二个参数，以相反的顺序迭代对象（按照键的降序排序）。

下列代码遍历了对象存储中的所有对象，按照键的降序排序：

```
var request = window.indexedDB.open("my-database", 4);

request.onsuccess = function(event) {
  var db = event.target.result;
  var exchangeTransaction = db.transaction("exchange_rates");
  var exchangeStore = exchangeTransaction.objectStore("exchange_rates");
  var exchangeCursor = exchangeStore.openCursor(null, "prev");
  exchangeCursor.onsuccess = function(event) {
    var cursor = event.target.result;
    if (!cursor) { return; }
    var rate = cursor.value;
    console.log(rate.exchange_from+" to "+rate.exchange_to+": "+rate.rate);
    cursor.continue();
  };
};
```

你会注意到，这次我们是在对象存储而不是索引上打开了游标，并且传递的第一个参数是 null 而不是 IDBKeyRange 对象，因为我们希望遍历对象存储中的所有对象。

在对象存储和索引上打开的游标，都可以接收以下参数：只有范围、只有方向、范围加上方向，或者两者都不传。

6.2.11 更新对象存储中的对象

当你知道一个对象的主键时，可以通过在对象存储中调用 put() 方法，传入要更新的对象和对象的主键，来快速更新它：

```
var request = window.indexedDB.open("my-database", 4);

request.onsuccess = function(event) {
  var updatedRate =
    {"exchange_from": "CAD", "exchange_to": "ILS", "rate": 1.2};
  var db = event.target.result;
  var exchangeTransaction = db.transaction("exchange_rates", "readwrite");
  var exchangeStore = exchangeTransaction.objectStore("exchange_rates");
  var request = exchangeStore.put(updatedRate, 2);
  request.onsuccess = function(event) {
    console.log("Updated");
  };
};
```

我们首先打开了一个 readwrite 事务，并获取了 exchange_rates 对象存储。随后，在对象存储上调用了 put() 方法，传入要更新的对象，以及想要替换的对象键名。

请注意，这只适用于使用 out-of-line key 的对象存储（参见 6.2.7 节中的“inline key 与 out-of-line key”），例如我们的 exchange_rates 对象存储（和 customers 对象存储相对，后者使用了 keypath，指向了客户的护照编号）。

当我们想在使用 inline key 的对象存储中更新对象，或者不知道对象的键名时，必须首先从对象存储中检索该对象。随后，可以通过在对象存储中调用 `put()`，或者在游标上调用 `update()`，来修改并更新数据存储。

下面的代码展示了这两种方法：

```
var request = window.indexedDB.open("my-database", 4);

request.onsuccess = function(event) {
  var db = event.target.result;
  var customerTransaction = db.transaction("customers", "readwrite");
  var customerStore = customerTransaction.objectStore("customers");
  var customerCursor = customerStore.openCursor();
  customerCursor.onsuccess = function(event) {
    var cursor = event.target.result;
    if (!cursor) { return; }
    var customer = cursor.value;
    if (customer.first_name === "Archie") {
      customer.first_name = "Archer";
      cursor.update(customer);
    } else {
      customer.first_name = "Tom";
      customerStore.put(customer);
    }
    cursor.continue();
  };
};
```

这段代码打开了一个遍历所有客户的游标，随后逐一检查了每个对象的名称。如果客户名是 `Archie`，我们就使用游标的 `update()` 方法将其修改为 `Archer`。否则，就使用对象存储的 `put()` 方法将其修改为 `Tom`。

注意，这次使用 `put()` 或者 `update()` 时，不需要指定每个对象的主键，因为我们传递的是原始对象（技术上来看，这是一份带有修改的克隆副本），其中已经包含了它的键名。

6.2.12　从对象存储删除对象

从对象存储删除对象的方式和修改对象十分相似。

下列代码会从 `exchange_rates` 对象存储中删除键名为 `2` 的对象：

```
var request = window.indexedDB.open("my-database", 4);

request.onsuccess = function(event) {
  var db = event.target.result;
  db.transaction("exchange_rates", "readwrite")
    .objectStore("exchange_rates")
    .delete(2);
};
```

如你所见，当你知道对象的键名，并且对象存储使用的是 out-of-line key 的时候，可以简单地在对象存储上调用 `delete()` 方法，传入对象的键名即可删除。

在所有其他情况下，你可以使用游标来迭代对象，并简单地在游标本身上调用 `delete()` 方法。这样将会删除游标当前指向的对象。

下面的代码将会遍历所有客户，并删除姓氏为 `Stevens` 的客户：

```
var request = window.indexedDB.open("my-database", 4);

request.onsuccess = function(event) {
  var db = event.target.result;
  db.transaction("customers", "readwrite")
    .objectStore("customers")
    .openCursor()
    .onsuccess = function(event) {
      var cursor = event.target.result;
      if (!cursor) { return; }
      var customer = cursor.value;
      if (customer.last_name === "Stevens") {
        cursor.delete();
      }
      cursor.continue();
    };
};
```

6.2.13　从对象存储中删除所有对象

你可以通过在对象存储上调用 `clear()`，来删除其中的所有对象。

和其他大多数 IndexedDB 操作类似，`clear()` 会返回一个请求，支持 `success` 和 `error` 事件。下列代码会清空 `customers` 对象存储，并在完成清空后，马上在控制台中打印信息：

```
var request = window.indexedDB.open("my-database", 4);

request.onsuccess = function(event) {
  var db = event.target.result;
  db.transaction("customers", "readwrite")
    .objectStore("customers")
    .clear()
    .onsuccess = function(event) {
      console.log("Object store cleared");
    };
};
```

6.2.14　处理冒泡IndexedDB错误

在 IndexedDB 中，错误事件会冒泡。

如果打开游标的请求抛出了错误，错误会被请求的 `onerror` 处理器捕获。但是，如果我们没有在那个请求上定义错误处理器，错误就会冒泡，被事务的错误处理器捕获。如果事务也没有定义错误处理器，那么错误就会再次冒泡，被数据库对象的错误处理器所捕获。

这种行为可以让你避免在每个请求或者事务上编写错误处理器。相反，你可以在数据库对象上编写一个错误处理器。

6.3 SQL忍者的IndexedDB

作为熟悉 SQL 的读者，我发现可以将 IndexedDB 的一些概念和熟悉的 SQL 概念进行比较，以便于掌握和记忆。

小心行事。大部分比较在最抽象的层面上是有意义的，但是当你仔细研究就会发现问题。它们就像那些给 PHP 开发者的 JavaScript 指南一样“正确”且实用——这是一种糟糕的学习方式，但有时候，你只是需要一个快速提醒，到底如何检查一个变量是 `empty()` 还是 `is_numeric()`。

如果你有 SQL 背景，不妨使用以下“备忘录”。

游标

打开游标和运行 `SELECT * FROM table;` 有点类似，它允许你获取整个对象，并对结果进行迭代。与 SQL 不同的是，游标只会指向对象，实际上没有返回对象（参见 6.2.6 节中的“游标是什么？”获取更多细节）。

IDBKeyRange

IDBKeyRange 对于游标的作用，相当于 `WHERE` 对于 `SELECT` 的作用。就像 `WHERE x = y` 可以让你将结果限制为只匹配 `y` 那样，`IDBKeyRange.only(y)` 也可以让游标只迭代对应的结果。类似地，`WHERE x >= y` 也可以表达为 `IDBKeyRange.lowerBound(y, false)`。

在 SQL 中 `WHERE` 可以查询任何列，而 IndexedDB 只允许你查询对象存储的索引，或者对象的键。

索引

在 SQL 中，索引可用来根据不同的列对数据库进行预索引，这样通过列的值进行表查询就可以更快地完成。IndexedDB 中的索引是一种更简化的形式，维护对象存储的索引，可以根据存储在其中的对象的单个属性来进行查询。

与 SQL 中可以对表格的任何一列进行查询（无论索引与否）不同，IndexedDB 只允许你使用已经索引的属性来限制游标。

游标方向

类似于 SQL 的 `ORDER BY x DESC`，在打开游标时，可以通过传递 `prev`，反转读取对象的顺序。和 SQL 不同的是，只能根据对象存储的键或者索引的键来排序（依赖于你在哪个对象上打开游标）。



《华盛顿邮报》——利用 IndexedDB 的离线分析

在构建新的渐进式 Web 应用时，《华盛顿邮报》的团队面临着一个有趣的挑战。在添加离线支持时，他们虽然提升了访问者的体验，却失去了测量和跟踪这些体验的能力。作为一个数据驱动的团队，这并不是他们愿意牺牲的。

直接与谷歌开发者关系小组的 Jeff Posnick 合作时，他们提出了一种解决方案：当 `fetch` 监听器捕获到失败的谷歌分析请求时，将其存储到 IndexedDB 中。随后，下一次 `fetch` 捕获到成功的谷歌分析请求时（意味着连接已经恢复），service worker 将重试所有失败的分析请求。

谷歌团队已经以辅助库的形式发布了这份代码，称之为 workbox-google-analytics，你可以将它用在自己的项目中。

6.4 IndexedDB实践

让我们将注意力转回哥谭帝国酒店应用。

这款应用跟踪了用户的预订，并显示在 My Account 页面。目前，这是通过从服务器获取 JSON 文件形式的预订数据来完成的，随后 service worker 将其缓存在 CacheStorage 中。每当用户操作这些数据（添加、修改或者删除预订）时，这份缓存的 JSON 文件就会过时。只有当用户再次请求页面时，才会从网络接收新的、有效的 JSON 文件，并替换缓存的版本。

这个案例中，客户端修改了数据之后，会使得客户端存储的数据失效，数据只能通过网络来更新。我们可以做得更好。预订数据可以作为 IndexedDB 的主要候选对象。

在 my-account.js 文件中，包含了驱动当前版本账号页面的逻辑：

```
$(document).ready(function() {

  // 请求并渲染用户预订内容
  populateReservations();

  // 添加预订控件的功能
  $("#reservation-form").submit(function(event) {
    event.preventDefault();
    var arrivalDate = $("#form--arrival-date").val();
    var nights = $("#form--nights").val();
    var guests = $("#form--guests").val();
    var id = Date.now().toString().substring(3, 11);
    if (!arrivalDate || !nights || !guests) {
      return false;
    }
    addReservation(id, arrivalDate, nights, guests);
    return false;
  });

  //定期检测未确认的预约
  setInterval(checkUnconfirmedReservations, 5000);
});

// 向服务端请求预订数据，并渲染到页面
var populateReservations = function() {
  $.getJSON("/reservations.json", renderReservations);
};

// 遍历未确认的预订，并向服务端验证其状态
var checkUnconfirmedReservations = function() {
  $(".reservation-card--unconfirmed").each(function() {
    $.getJSON(
      "/reservation-details.json",
      {id: $(this).data("id")},
      function(data) {
```

```
        updateReservationDisplay(data);
      });
    });
  };

  // 添加一个等待状态的预订到DOM结构中，并尝试向服务端发起预订
  var addReservation = function(id, arrivalDate, nights, guests) {
    var reservationDetails = {
      id:           id,
      arrivalDate:  arrivalDate,
      nights:       nights,
      guests:       guests,
      status:       "Awaiting confirmation"
    };
    renderReservation(reservationDetails);
    $.getJSON("/make-reservation", reservationDetails, function(data) {
      updateReservationDisplay(data);
    });
  };
```

这段脚本相对简单，实现了以下功能。

(1) 调用 `populateReservations()`，它从服务器加载 reservations.json，逐一遍历结果，并将结果添加到 DOM 中（使用 `renderReservations` 函数）。
(2) 向预订按钮添加验证表单数据的逻辑，然后渲染新的预订内容到 DOM 中，并向服务器发送新的预订。
(3) 每五秒调用一次 `checkUnconfirmedReservations` 函数，检查未确认的预订，并通过联系服务器查看其状态是否更新。

文件的剩余部分（未在前面的代码中显示）包含了 `renderReservations()`、`renderReservation()` 和 `updateReservationDisplay()` 方法的定义，这些方法会接收预订细节，并将其渲染到 DOM 中。我们不会覆盖或者修改这部分内容。



这段脚本可以通过很多方式进行改进，从它依赖 DOM 中的数据作为真实来源、不断轮询网络进行更新的方式，到处理错误（或者忽略错误）的方式。实际上，我们故意保持代码简单，以便专注于本章的核心概念。

我们将会分两个阶段来升级到 IndexedDB。首先，修改代码，将网络请求的所有预订存储到本地数据库中。我们的修改版 `populateReservations()` 方法，将总是试图从数据库读取预订数据，只有当本地数据不存在时，才回退到网络。其次，我们将修改添加预订的代码，以及修改定期从网络获取预订状态的代码。这两者都会被修改成保持本地数据库的数据最新，并且与服务端同步。

一如往常，首先要确保你的代码处于上一章结束时的状态。为此，在命令行中运行以下命令：

```
git reset --hard
git checkout ch06-start
```

在项目的 public/js 目录中，添加一个空文件，并命名为 reservations-store.js。这个文件将会包含我们的 IndexedDB 代码。

接下来，我们要确保账号页面会加载这个文件。在 my-account.html 接近末尾的地方，添加一个 `<script>` 标签，放在 `app.js` `<script>` 标签的上方：

```
<script src="/js/reservations-store.js"></script>
<script src="/js/app.js"></script>
<script src="/js/my-account.js"></script>
```

为确保用户在离线时也能访问他们的预订，打开 serviceworker.js，并添加 `/js/reservations-store.js` 到 `CACHED_URLS` 数组中。

现在开始编写 IndexedDB 代码。在 reservations-store.js 中，添加下列代码：

```
var openDatabase = function() {
  // 在尝试使用IndexedDB之前，需要确保浏览器支持IndexedDB
  if (!window.indexedDB) {
    return false;
  }

  var request = window.indexedDB.open("gih-reservations", 1);

  request.onerror = function(event) {
    console.log("Database error: ", event.target.error);
  };

  request.onupgradeneeded = function(event) {
    var db = event.target.result;
    if (!db.objectStoreNames.contains("reservations")) {
      db.createObjectStore("reservations",
        { keyPath: "id" }
      );
    }
  };

  return request;
};

var openObjectStore = function(storeName, successCallback, transactionMode) {
  var db = openDatabase();
  if (!db) {
    return false;
  }
  db.onsuccess = function(event) {
    var db = event.target.result;
    var objectStore = db
      .transaction(storeName, transactionMode)
      .objectStore(storeName);
    successCallback(objectStore);
  };
  return true;
};

var getReservations = function(successCallback) {
  var reservations = [];
  var db = openObjectStore("reservations", function(objectStore) {
    objectStore.openCursor().onsuccess = function(event) {
```

```
        var cursor = event.target.result;
        if (cursor) {
          reservations.push(cursor.value);
          cursor.continue();
        } else {
          if (reservations.length > 0) {
            successCallback(reservations);
          } else {
            $.getJSON("/reservations.json", function(reservations) {
              openObjectStore("reservations", function(reservationsStore) {
                for (var i = 0; i < reservations.length; i++) {
                  reservationsStore.add(reservations[i]);
                }
                successCallback(reservations);
              }, "readwrite");
            });

          }
        }
      };
    });
    if (!db) {
      $.getJSON("/reservations.json", successCallback);
    }
  };
```

我们的新代码中首先定义了一些实用的函数来处理数据库。

第一个函数 `openDatabase()` 会打开一个新的数据库请求，设置基本的错误日志，定义数据库的更新方法，并在方法中创建 `reservations` 对象存储。如果浏览器不支持 IndexedDB，它就会返回 `false`，否则就会返回请求对象。由于返回的请求对象没有包含 `onsuccess` 事件，所以我们后续可以这样使用它：

```
var db = openDatabase().onsuccess = function(event) {}
if (!db) { console.log("IndexedDB not supported"); }
```

第二个函数 `openObjectStore()` 会在对象存储上打开一个事务，并在其上运行函数。它会接受对象存储的名称作为第一个参数，打开成功时运行的回调函数作为第二个参数，以及一个可选的第三参数，其中包含了需要打开的事务类型——`readonly`（默认）或者 `readwrite`。如果浏览器支持 IndexedDB，函数会返回 `true`，否则返回 `false`。使用这个函数的一个简单示例是：

```
var db = openObjectStore("reservations", function(objectStore) {
  objectStore.openCursor().onsuccess = function() {};
}, "readwrite");
if (!db) { console.log("IndexedDB not supported"); }
```

最后我们创建了 `getReservations()` 函数。该函数接收一个回调函数，在执行回调时传入一个包含了所有用户预订的数组。这些预订数据会从本地 IndexedDB 数据库或者从服务端返回。函数首先打开 `reservations` 对象存储，创建游标，并对所有数据进行迭代（见图 6-2）。在探索游标的时候（参见 6.2.6 节），我们看到游标的 `onsuccess` 函数，在游标每次前进到一个新的记录时都会被调用，甚至在游标通过最后一条记录之后也会被调用（如

果对象存储为空，就会发生在第一次调用 `onsuccess` 回调的时候）。基于这个原因，我们在 `onsuccess` 回调的一开始需要确保游标是指向某条记录的。如果这个判断为真，我们就将记录放入预订数组中，并将游标向前移动。如果游标没有指向任何内容（要么是因为对象存储为空，要么是因为游标通过最后一条记录），我们就会查看预订数组。如果预订数组不为空，此时我们就在数组中获得了所有的预订数据，随后调用 `successCallback`，传入这个数组。如果在遍历对象存储中的每条记录之后，预订数组依然为空，那么我们就会通过向网络请求 `reservations.json` 来检索它。当接收到 JSON 数据后，我们对其进行迭代，将每条预订添加到对象存储中。一旦所有预订数据存储到 IndexedDB 之后，我们就调用 `sucessCallback`，传入预订数组。在函数末尾，我们检查了是否支持 IndexedDB。如果浏览器不支持 IndexedDB，那么 `openObjectStore` 就会立即返回 `false`，从而触发最后一个条件，从网络中获取预订数据作为代替。

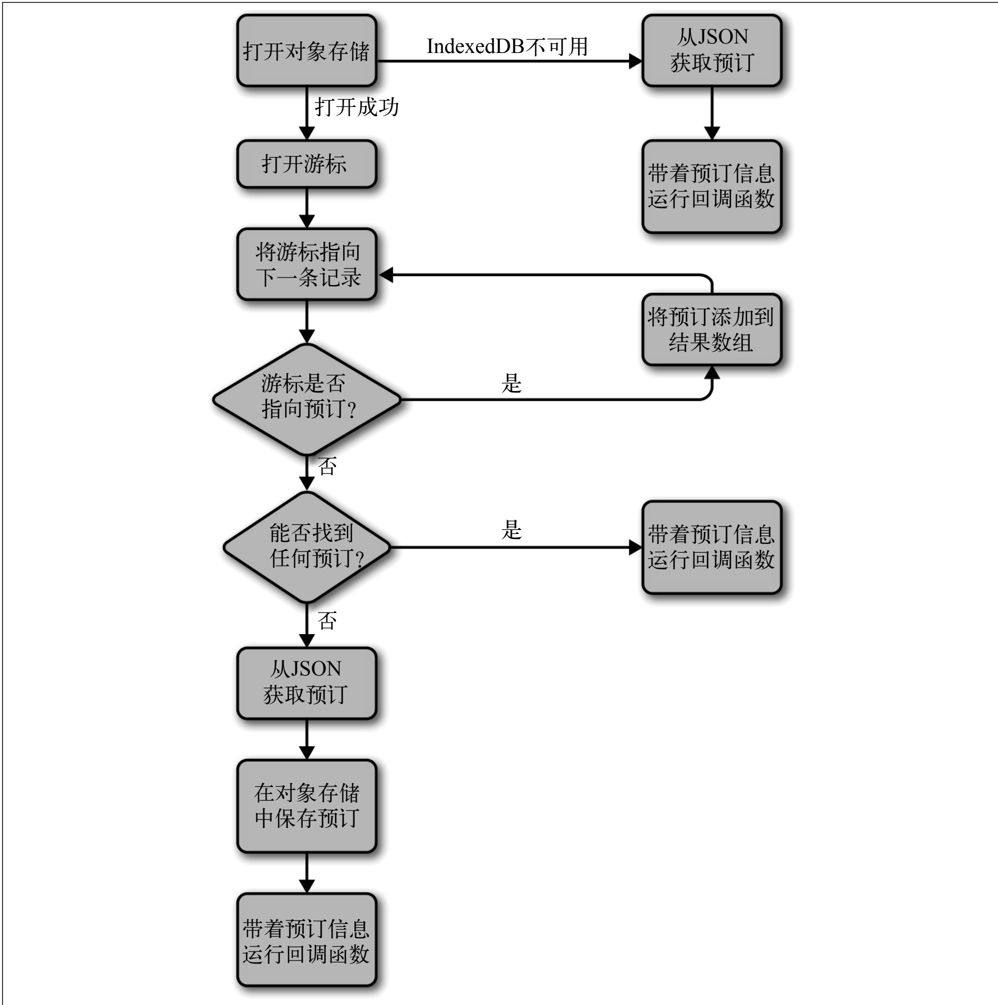


图 6-2：`getReservations()` 逻辑的流程图



在 Firefox 中，当用户禁用第三方 cookie 时，尝试使用 IndexedDB 会抛出错误。如果你想让代码在这样的环境中运行（例如，当用户特别禁用了第三方 cookie，而代码尝试在第三方网站中嵌入的 iframe 中运行），可能你需要使用 `try...catch` 语句，将 `window.indexedDB` 的所有调用封装起来。

现在，用于存储和访问 IndexedDB 中预订的框架已经到位，是时候投入使用了。

在 my-account.js 中，现有的 `populateReservations` 函数代码是这样的：

```
var populateReservations = function() {
  $.getJSON("/reservations.json", renderReservations);
};
```

这个函数中调用了 `$.getJSON()`，获取一个预订对象的数组，并传递给一个回调函数。我们可以设计一个 `getReservations` 函数，使其同样接收并调用一个回调函数，并传入一个类似结构的数组，从而使得这两个函数可以互换。

将 `populateReservations` 函数替换成下列代码：

```
var populateReservations = function() {
  getReservations(renderReservations);
};
```

当你下次访问页面时，将会创建一个 IndexedDB 数据库，获取 reservations.json 中的内容，并存储起来，随后 DOM 结构会使用本地数据库中的预订数据进行更新。如果你再次刷新页面，就会显示同样的数据，但是页面不会发起 reservations.json 的请求。数据是直接从本地数据库加载的。

如果用户要创建新的预订，或者其中一个预订的状态发生了变化，会怎么样呢？目前，一旦预订数据检索并存储到本地数据库之后，数据就不会发生变化了。让我们来解决这个问题。

在 reservations-store.js 中，将以下代码添加到 `getReservations()` 的定义之前：

```
var addToObjectStore = function(storeName, object) {
  openObjectStore(storeName, function(store) {
    store.add(object);
  }, "readwrite");
};

var updateInObjectStore = function(storeName, id, object) {
  openObjectStore(storeName, function(objectStore) {
    objectStore.openCursor().onsuccess = function(event) {
      var cursor = event.target.result;
      if (!cursor) { return; }
      if (cursor.value.id === id) {
        cursor.update(object);
        return;
      }
      cursor.continue();
    };
  }, "readwrite");
};
```

第一个新函数接收对象存储的名称以及要放进存储的新对象作为参数。这个函数可以这样调用：

```
addToObjectStore("reservations", { id: 123, nights: 2, guests: 2 });
```

第二个函数接收对象存储的名称，找到与给定 `id` 参数匹配的 `id` 的对象，并用它更新新对象。这是通过在对象存储上打开 `readwrite` 事务，并使用游标进行迭代来完成的。在游标到达最后一条记录或者匹配成功之前，函数会一直迭代。如果找到匹配项，就会通过调用 `cursor.update(object)` 进行更新。此时，函数会通过 `return` 退出执行，因为一旦找到匹配，就不需要继续迭代下一条记录了。这个函数可以这样调用：

```
updateInObjectStore("reservations", 123, { id: 123, nights: 5, guests: 1 });
```

最后一步是在 IndexedDB 中添加或更新数据时调用这两个函数。

在 my-account.js 中修改 `addReservation` 方法，让其在添加新预订到服务器之前，先调用 `addToObjectStore()`。修改后的函数应该如下所示：

```
var addReservation = function(id, arrivalDate, nights, guests) {
  var reservationDetails = {
    id:          id,
    arrivalDate:  arrivalDate,
    nights:       nights,
    guests:       guests,
    status:       "Awaiting confirmation"
  };
  addToObjectStore("reservations", reservationDetails);
  renderReservation(reservationDetails);
  $.getJSON("/make-reservation", reservationDetails, function(data) {
    updateReservationDisplay(data);
  });
};
```

在 my-account.js 中修改 `checkUnconfirmedReservations` 方法，无论服务端是否返回新的数据，都调用 `updateInObjectStore` 方法。修改后的函数应该如下所示：

```
var checkUnconfirmedReservations = function() {
  $(".reservation-card--unconfirmed").each(function() {
    $.getJSON(
      "/reservation-details.json",
      {id: $(this).data("id")},
      function(data) {
        updateInObjectStore("reservations", data.id, data);
        updateReservationDisplay(data);
      }
    );
  });
};
```

6.5 promise式的数据库

现在你已经对 IndexedDB 略知一二了，可能你会开始注意到它的缺点。作为一个早于 promise 提出的 API，IndexedDB 在很大程度上依赖于回调——常言道，通往地狱的路就是由回调函数铺平的。

让我们看看以下代码，使用回调函数来更新 IndexedDB 中的对象：

```
var request = window.indexedDB.open("gih-reservations", 1);

request.onerror = function(event) {
  console.log("Database error: ", event.target.error);
};

request.onsuccess = function(event) {
  var db = event.target.result;
  var objectStore = db
    .transaction("reservations", "readwrite")
    .objectStore("reservations");

  var request = objectStore.add({id:1, rooms: 1, guests: 2});
  request.onsuccess = function(event) {
    console.log("Object added");
  };
  request.onerror = function(event) {
    console.log("Database error: ", event.target.error);
  };
};
```

我们打开了一个打开数据库的请求，然后把回调附加到该请求上。在回调中，我们又进一步请求了事件，再一次附加回调，以此类推。实际上，这个示例代码只是使用 IndexedDB 的一个很简单的例子。你打开的请求越多，代码量就越大，长此以往，应用就会陷入到俗称的**回调地狱**中。

现在，我们来考虑使用 promise 式的 IndexedDB 语法作为替代。代码可能是这样的：

```
openDatabase("gih-reservations", 1).then(function(db) {
  return openObjectStore(db, "reservations", "readwrite");
}).then(function(objectStore) {
  return addObject(objectStore, {id: 1, rooms: 1, guests: 2});
}).then(function() {
  console.log("Object added");
}).catch(function(errorMessage) {
  console.log("Database error: ", errorMessage);
});
```

这种方式编写的代码可读性要好得多，并且我们可以轻易地扩展它，而不会陷入到回调地狱中。

幸运的是，JavaScript 让我们可以轻松地将使用回调的异步代码转换成 promise。

在试图构建基于 promise 的 IndexedDB 替代方案之前，我们先来看看如何将一个简单的异步 API 转换成基于 promise 的 API：

```
var request = new XMLHttpRequest();

request.onload = function() {
  // 处理响应
};

request.onerror = function() {
  // 处理错误
}

request.open("get", "/events.json", true);
request.send();
```

这段代码是旧式的异步 XMLHttpRequest 代码，依赖于回调。

如何将其变成基于 promise 的 API？这个逻辑看起来应该如下所示：

```
在请求一个基于promise的XMLHttpRequest时：
  创建一个新promise
  在promise中运行：
    var request = new XMLHttpRequest();
    当request.onload被调用时，调用promise的resolve事件
    当request.onerror被调用时，调用promise的reject事件
    将XMLHttpRequest发送到互联网的某处
  返回promise
```

在 JavaScript 中，代码看起来就像这样：

```
var promised_XMLHttpRequest = function(url, method) {
  return new Promise(function(resolve, reject) {
    var request = new XMLHttpRequest();
    request.onload = resolve;
    request.onerror = reject;
    request.open(method, url, true);
    request.send();
  });
};
```

这个新的 `promised_XMLHttpRequest` 函数，接收一个 `url` 和 `method`，并返回一个 `new Promise`。这个新的 promise 中传入了一个回调函数，其中包含了我们的 XMLHttpRequest 代码。要记住，这个回调函数同时包含了 `resolve` 和 `reject` 参数，我们可以在完成或者拒绝 promise 时，分别调用这两个函数其中之一。在新的 XMLHttpRequest 代码中，当 XMLHttpRequest 的 `onload` 回调执行时，我们完成 promise，当 XMLHttpRequest 的 `onerror` 回调执行时，我们拒绝 promise。

换句话说，在 promise 中的 XMLHttpRequest 代码依然是旧式的回调风格代码。但是我们将 promise 的回调赋值给 XMLHttpRequest，让它可以跟 promise 交互。

然后，我们就可以调用新的 `promised_XMLHttpRequest()` 函数，并像使用 promise 那样，与它进行交互：

```
promised_XMLHttpRequest("/events.json", "get").then(function() {
  // 处理响应
}).catch(function() {
```

```
  // 处理错误
});
```

使用相同的方法，我们就可以用 promise 包装不同的 IndexedDB 函数，并创建出 `openDatabase()`、`openObjectStore()` 和 `addObject()` 函数：

```
var openDatabase = function(dbName, dbVersion) {
  return new Promise(function (resolve, reject) {
    if (!window.indexedDB) {
      reject("IndexedDB not supported");
    }

    var request = window.indexedDB.open(dbName, dbVersion);

    request.onerror = function(event) {
      reject("Database error: " + event.target.error);
    };

    request.onupgradeneeded = function(event) {
      // 更新的代码
    };

    request.onsuccess = function(event) {
      resolve(event.target.result);
    };
  });
};

var openObjectStore = function(db, storeName, transactionMode) {
  return new Promise(function (resolve, reject) {
    var objectStore = db
      .transaction(storeName, transactionMode)
      .objectStore(storeName);
    resolve(objectStore);
  });
};

var addObject = function(objectStore, object) {
  return new Promise(function (resolve, reject) {
    var request = objectStore.add(object);
    request.onsuccess = resolve;
  });
};
```

现在，我们拥有了 IndexedDB 基于 promise 的 API，可以用它来更加优雅地访问数据库：

```
openDatabase("gih-reservations", 1).then(function(db) {
  return openObjectStore(db, "reservations", "readwrite");
}).then(function(objectStore) {
  return addObject(objectStore, {id:1, rooms: 1, guests: 2});
}).then(function() {
  console.log("Object added");
}).catch(function(errorMessage) {
  console.log("Database error: ", errorMessage);
});
```

结合迄今为止学到的一切，我们就可以使用 promise，重写哥谭帝国酒店的 IndexedDB 代码了。这样我们就可以在下一章中使用更加简单的方式来访问数据库。

将 reservations-store.js 中的内容修改成下列代码：

```
var DB_VERSION = 1;
var DB_NAME = "gih-reservations";

var openDatabase = function() {
  return new Promise(function(resolve, reject) {
    // 在使用IndexedDB之前，要确保它是被支持的
    if (!window.indexedDB) {
      reject("IndexedDB not supported");
    }
    var request = window.indexedDB.open(DB_NAME, DB_VERSION);
    request.onerror = function(event) {
      reject("Database error: " + event.target.error);
    };

    request.onupgradeneeded = function(event) {
      var db = event.target.result;
      if (!db.objectStoreNames.contains("reservations")) {
        db.createObjectStore("reservations",
          { keyPath: "id" }
        );
      }
    };

    request.onsuccess = function(event) {
      resolve(event.target.result);
    };
  });
};

var openObjectStore = function(db, storeName, transactionMode) {
  return db
    .transaction(storeName, transactionMode)
    .objectStore(storeName);
};

var addToObjectStore = function(storeName, object) {
  return new Promise(function(resolve, reject) {
    openDatabase().then(function(db) {
      openObjectStore(db, storeName, "readwrite")
        .add(object).onsuccess = resolve;
    }).catch(function(errorMessage) {
      reject(errorMessage);
    });
  });
};

var updateInObjectStore = function(storeName, id, object) {
  return new Promise(function(resolve, reject) {
    openDatabase().then(function(db) {
      openObjectStore(db, storeName, "readwrite")
        .openCursor().onsuccess = function(event) {
```

```
        var cursor = event.target.result;
          if (!cursor) {
            reject("Reservation not found in object store");
        }
          if (cursor.value.id === id) {
            cursor.update(object).onsuccess = resolve;
            return;
        }
          cursor.continue();
        };
    }).catch(function(errorMessage) {
      reject(errorMessage);
    });
  });
};

var getReservations = function() {
  return new Promise(function(resolve) {
    openDatabase().then(function(db) {
      var objectStore = openObjectStore(db, "reservations");
      var reservations = [];
      objectStore.openCursor().onsuccess = function(event) {
        var cursor = event.target.result;
        if (cursor) {
          reservations.push(cursor.value);
          cursor.continue();
        } else {
          if (reservations.length > 0) {
            resolve(reservations);
          } else {
            getReservationsFromServer().then(function(reservations) {
              openDatabase().then(function(db) {
                var objectStore =
                  openObjectStore(db, "reservations", "readwrite");
                for (var i = 0; i < reservations.length; i++) {
                  objectStore.add(reservations[i]);
                }
                resolve(reservations);
              });
            });
          }
        }
      };
    }).catch(function() {
      getReservationsFromServer().then(function(reservations) {
        resolve(reservations);
      });
    });
  });
};

var getReservationsFromServer = function() {
  return new Promise(function(resolve) {
    $.getJSON("/reservations.json", resolve);
  });
};
```

这份新代码使用了本节前面介绍的技术，让我们得以修改函数并返回 promise。它还把从服务端获取预订数据的代码提取到了一个名为 `getReservationsFromServer()` 的新函数中，并返回一个 promise。



我们唯一没有修改成返回 promise 的函数是 `openObjectStore()`。在 Firefox 中，promise 打开的事务会在 promise 的 `resolve` 运行之前完成。换句话说，当我们试图在 promise 中打开对象存储的时候，对象存储的事务已经关闭了。

修改后的 `getReservations()` 函数也会返回 promise。它封装了所有的游标遍历，而不是将其暴露给代码的其余部分，并且只有在完成对象存储中所有条目的遍历，并从此构建出一个新数组之后，才完成这个 promise。

无论我们从 IndexedDB 还是从服务端获取预订数据，`getReservations()` 都会把所有的异步复杂性隐藏到一个友好的 promise 接口中，然后我们可以在 `populateReservations()` 中使用它：

```
var populateReservations = function() {
  getReservations().then(function(reservations) {
    renderReservations(reservations);
  });
};
```

可以看到，`then` 接收了一个只接收一个参数的函数，并且调用了另外一个只接收一个参数的函数。我们可以进一步简化代码，直接把这个函数传递给 `then`：

```
var populateReservations = function() {
  getReservations().then(renderReservations);
};
```

在 my-account.js 中，修改 `populateReservations()` 函数，改为使用新的基于 promise 的 `getReservations()` 函数，如上述的代码片段所示。

6.6 IndexedDB管理

和缓存一样，当你在 IndexedDB 中存储越来越多的数据时，需要考虑在用户设备上占用的存储量。

对于哥谭帝国酒店应用来说，这不太可能成为问题，用户它使用的数据量增长缓慢，而且呈线性。但是，让我们在不同上下文中考虑 IndexedDB 的情况——我们的消息应用。

应用可以将所有从服务端接收到的数据存储到 IndexedDB，从本地数据库填充接口数据，取代网络方案。它甚至允许用户在离线时编写新的消息（或许是将消息保存在一个未发送消息的对象存储中）。通过采用这种方法，我们可以实现一个在离线情况下也能拥有部分在线功能的应用，唯一不同的只是内容的新鲜程度。

但是，在 IndexedDB 中保存所有的消息将会占用用户设备越来越多的内存空间。最终，我们可能会达到浏览器分配给我们的存储限制。在构建应用时，负责任的做法是先从对象存储中删除旧的消息，只保留最新消息。其中一种做法，是在消息的发布日期上使用索引，

通过索引获取消息，并删除所有旧于特定天数的消息。另外一种办法，是保留最新的 100 条消息，并删除所有更旧的消息。

无论你选择哪种方式，始终考虑在用户设备上占用的空间，并负责任地采取行动，才是最重要的。请记住，当到达某个存储上限时，你在用户设备上存储的任何数据都可能会被清除。有关存储限制的详细信息，请参见 4.5 节中的“存储限制”。



如果你想要确保保存的数据不会被自动删除，在 Chrome 和 Opera 中，可以使用新的实验性 API，向设备请求持久化存储的权限：

```
if (navigator.storage && navigator.storage.persist) {
  navigator.storage.persist().then(function(granted) {
    if (granted) {
      console.log("Data will not be deleted automatically");
    }
  });
}
```

一旦授权后，存储的任何内容将不会被设备自动删除。内容只能通过用户操作进行删除。

6.7 在service worker中使用IndexedDB

在第 7 章中，我们需要从 service worker 访问 `reservations` 对象存储。幸运的是，在 service worker 中访问 IndexedDB 的方式，和通过页面访问的方式完全一样。

为了避免重写所有我们辛辛苦苦写好的 IndexedDB 代码，需要确保 reservations-store.js 可以在 service worker 中正常工作，就像在页面上一样。

要做到这一点，需要实现两件事。

首先，我们的代码使用 `window.indexedDB` 来调用 IndexedDB API。然而在 service worker 中无法访问 `window` 对象。它是在一个完全不同的上下文中运行的。service worker 可以通过它可访问的全局对象来访问 IndexedDB。

为了编写可以同时在 service worker 和页面中运行的代码，可以使用 `self.indexedDB`。在 service worker 中，`self` 会指向全局对象，而在页面中，`self` 会指向 `window`。

在 reservations-store.js 中，将每一个 `window.indexedDB` 的调用，修改为 `self.indexedDB`（应该只有两处需要修改）。

接下来是我们应用特有的一个修改。在 reservations-store.js 的末尾是 `getReservationsFromServer()` 的代码。现有的代码是这样的：

```
var getReservationsFromServer = function() {
  return new Promise(function(resolve, reject) {
    $.getJSON("/reservations.json", resolve);
  });
};
```

你能发现其中的问题吗？

我们在此处处理的代码依赖于 jQuery 函数 `$.getJSON` 来获取 reservations.json 文件。不幸的是（或者说，幸运的是），在 service worker 中我们没有引入 jQuery，所以调用 `$.getJSON` 会导致报错。我们可以使用 `fetch()` 来替换这段代码，它可以同时在页面和 service worker 中工作，不过，在旧的浏览器中 fetch 可能不能正常工作。由于我们不打算放弃这部分用户，所以会在代码中同时包含 `fetch()` 和 `$.getJSON`，并使用功能检测来判断哪一个是可用的。

在 reservations-store.js 中，将 `getReservationsFromServer()` 的代码替换为如下代码：

```
var getReservationsFromServer = function() {
  return new Promise(function(resolve) {
    if (self.$) {
      $.getJSON("/reservations.json", resolve);
    } else if (self.fetch) {
      fetch("/reservations.json").then(function(response) {
        return response.json();
      }).then(function(reservations) {
        resolve(reservations);
      });
    }
  });
};
```

这段代码首先检测了 `self`（window 或者 service worker 的全局对象）中是否可以使用 `$`（即 jQuery）。如果可以，代码就使用 `$.getJSON` 来获取 JSON，并完成 promise。否则，代码会检查 `fetch` 是否可用，如果可用就使用它来获取 JSON。

当我们调用 `fetch("/reservations.json")` 时，会得到一个 promise，其中包含了响应对象。由于响应对象中包含了 JSON，我们可以使用对象的 `json` 方法，得到解析后的 JSON 数据（当然，这是包含在 promise 中的）。随后，我们可以使用 JSON 创建出的对象，完成我们的 promise。

现在，reservations-store.js 文件已经准备好在 service worker 中使用了。

在 serviceworker.js 文件的顶部，添加这行代码：

```
importScripts("/js/reservations-store.js");
```

在 service worker 中，`importScripts` 是可以用来加载脚本的特殊方法。

6.8 IndexedDB生态系统

开源社区已经出现了许多 IndexedDB 相关的库，致力于使 IndexedDB 更易用。其中的一些库通过使用 promise 替代回调式的代码，让 IndexedDB 的使用变得更加优雅，另一些库则专注于提高跨浏览器的兼容性，或者是简化浏览器和服务器之间的数据同步。

下面介绍四个比较受欢迎的库。

6.8.1 PouchDB

PouchDB 的创建目标是在浏览器中运行一个 JavaScript 数据库，让应用可以在离线时在本

地存储数据。

PouchDB 受到了 CouchDB 数据库的启发，两者可以轻易地结合在一起，让你的应用可以在浏览器和服务器之间同步数据。

PouchDB 优先使用 IndexedDB，如果不支持 IndexedDB 或者缺少某些功能，则使用 Web SQL 作为回退（这是一个老式的、被抛弃的 API，在许多浏览器中依然支持）：

```
var db = new PouchDB("reservations-db");

db.put({
  _id: 1,
  nights: 3,
  guests: 2
});

db.changes().on("change", function() {
  console.log("Reservations database changed");
});

db.replicate.to("https://db.gothamimperial.com/mydb");
```

6.8.2 localForage

localForage 是一个使用 localStorage 风格 API（同时支持回调和 promise）的浏览器 JavaScript 数据库，可以用来简化离线应用的创建。

它依赖于 IndexedDB 或 WebSQL，并在旧浏览器中回退到 localStorage：

```
var id = 1;
localforage
  .setItem(id, { nights: 3, guests: 2 })
  .then(function() {
    return localforage.getItem(id);
  })
  .then(function(reservation) {
    console.log("Reservation "+id+" is for "+reservation.nights+" nights");
  });
```

6.8.3 Dexie.js

Dexie.js 包装了 IndexedDB，通过许多方式提升了 IndexedDB 的开发体验，包括优雅的 API、更简单的查询，以及改进的错误处理：

```
var db = new Dexie("reservations");

// 定义一个schema
db.version(1).stores({
  reservations: "++id, ,nights, guests"
});

db.open();
```

```
db.reservations
  .where("guests")
  .above(8)
  .each(function (reservation) {
    console.log(
      "Reservation " + reservation.id + " for " + reservation.nights + " nights"
    );
  });
```

6.8.4 IndexedDB Promised

IndexedDB Promised 是一个轻量的包装库，其目的是改进 IndexedDB 的使用体验。只需要简单一句话就可以概括它的功能："Promise 风格的 IndexedDB"。

```
idb.open("reservations", 1, function(upgradeDB) {
  return upgradeDB.createObjectStore("reservations");
}).then(function(db) {
  return db.transaction("reservations").objectStore("reservations").get(1);
}).then(function(reservation) {
  console.log("Reservation for " + reservation.nights + " nights");
});
```

6.9 小结

通过使我们的应用能够在本地数据库中存储、修改并访问数据，我们成功实现了切断服务器依赖的最后一步。

通过组合 service worker、缓存和本地数据库，我们最终可以构建一个与用户连接状态无关的渐进式 Web 应用。这类应用可以在毫秒级别内加载，显示并操作内容和数据。和原生应用类似，仅只要当我们想从服务器中检索更新的数据和内容，或者就用户操作与服务器通信时，才需要网络连接。

哥谭帝国酒店的客户可以随时随地访问他们的账号页面，无论他们的网络连接状况如何。他们可以看到预订的状态和详情，并查看酒店即将举行的活动。

不仅如此，我们还可以更进一步，让用户在离线状态下也能够发起新的预订。试想，用户不仅可以在离线状态下看到内容和数据，还可以进行操作，并在下一次上线时将这些操作同步给服务器。

要实现这一点，一种方法是在 IndexedDB 中存储离线的预订，并添加一段脚本在页面中定时运行，将所有在 IndexedDB 中找到的离线预订添加到服务器。但是，这种方法要求用户一直打开应用，直到连接恢复后，操作才能完成。

在第 7 章中，我们将研究一种最令人兴奋的新技术，让用户即使在离线时也能进行操作，并且一旦连接恢复，这些操作就能继续进行，甚至在用户关闭浏览器之后也是如此。

第 7 章

使用后台同步保证离线功能

对用户来说，没有什么比填写表单，点击提交按钮，然后得到一个连接错误响应更加令人沮丧的了。填写表单是一个缓慢且沮丧的过程，尤其是在移动设备上——因为在不恰当的时刻进了电梯，然后所有的努力都白费了，这让很多用户非常沮丧。

缓慢、不可靠的连接同样令人沮丧。如果我们点击了网站上的某个按钮，等待结果发生，然后一旦我们不想再等了，试图在操作完成结束之前跳转页面，会发生什么呢？操作可能会在我们不知情的情况下完成，也可能不会完成。作为开发者，我们不得不求助于一些技术，诸如监听页面的 `onbeforeunload` 事件，然后显示一条信息，恳求用户再等一会（实际显示的按钮是 OK/ 取消——坦白说，我记不清其中的哪个按钮意味着等待了，具体参见图 7-1）。

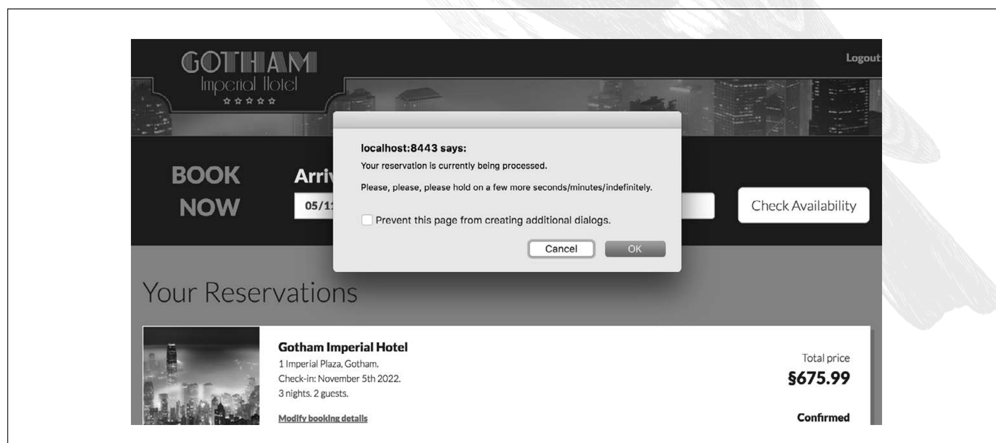


图 7-1：退出确认

作为用户，我们不能接受那些时不时会抹去所有辛勤工作的软件。每当附近的信号塔负载太高，我们都期望软件和移动应用不要这样对待我们。然而不幸的是，这依然是试图在移动 Web 上完成工作时所面临的现实。

这种固有的不可靠性一直是 Web 与原生应用的主要不同。现在，一项称为**后台同步**（background sync）的新技术，终于让我们可以对此做些什么了。

后台同步使得我们能够确保用户采取的任何操作都能完成（无论是填写表单、点击“请回复”按钮，还是发送消息），不管用户的连接状态如何。甚至即使用户离开我们的 Web 应用，不再回来，并关闭浏览器，后台同步操作也能够完成。这是浏览器近年来给我们带来的最有价值的工具之一。它可能不像消息推送、主屏图标甚至是离线功能那般耀眼。如果实现恰当，它的影响对于用户是不可见的。但是，正因同步功能在后台不知疲倦地工作，用户的任务才能得以完成。

对于用户而言，能够信任你的渐进式 Web 应用总是能够工作（不仅是有时能工作，并且不依赖于连接、接收或者天气状况），意味着传统的 Web 应用和这种应用是有差异的，后者能够实现原生应用般的效果。

对于企业来说，让用户在连接失败时也能够订票、订阅新闻或者发送消息时，对他们的底线也会产生积极的影响。

后台同步作为一种低调的、相对简单的实现方式，是现代渐进式 Web 应用的核心组件，也是“离线优先”的最后一个要素。

7.1　后台同步是如何工作的

使用后台同步的实质，是将操作从页面上下文中剥离开，并在后台运行。

通过将这些操作放到后台，它们就不会受到单个网页行为不可预测性的影响。网页会被关闭，用户连接可能会断开，甚至服务器有时候也会故障。但是，只要用户设备上安装了浏览器，后台同步中的操作就不会消失，直到它成功完成为止。

你应该考虑用后台同步来处理任何超出当前页面生命周期的操作。无论用户要发送消息、将待定项目标记为已完成，还是添加事件到日历，后台同步都可以确保操作能够成功完成。

使用后台同步很简单，不是在页面上直接执行操作（例如 Ajax 调用），而是注册一个同步事件：

```
navigator.serviceWorker.ready.then(function(registration) {
  registration.sync.register('send-messages');
});
```

这段代码可以在页面中运行。它获取了当前激活 service worker 的 registration 对象，并用其注册了一个称为 `send-messages` 的 sync 事件。

接下来，你可以将一个监听该同步事件的事件监听器添加到 service worker 中。这个事件包含的逻辑将会在 service worker 中执行，而不是在页面上：

```
self.addEventListener("sync", function(event) {
  if (event.tag === "send-messages") {
    event.waitUntil(function() {
      var sent = sendMessages();
      if (sent) {
        return Promise.resolve();
      }else{
        return Promise.reject();
      }
    });
  }
});
```

注意，事件监听器的代码使用 `waitUntil` 来确保我们可以控制事件结束的时机。这使得我们有时间尝试执行某些操作，如果操作成功才完成事件，否则就拒绝。如果我们给 sync 事件返回了拒绝的 promise，浏览器就会将同步操作放入队列，并在稍后重试。这个名为 `send-messages` 的同步事件会一直重试直到成功，甚至是在用户已经离开应用的情况下。



让我们再次回到示例消息应用，看看它如何从后台同步功能中获益。

要让我们的消息应用取得成功，必须要让用户感觉到应用是可信赖的。用户应该可以随时打开应用，写下他们的想法，点击提交，然后继续他们的生活。用户在编写消息之前，不需要担心连接状态。用户永远不应该被错误消息拒之门外，然后被要求重试。连接丢失是我们必须计划在内的事情，以便可以优雅地处理它。如果这种情况处理得不好，就会破坏用户对应用的信任。

WhatsApp 的原生应用完美示范了这一点。你可以随时打开它（不管连接状态如何），编写消息，并知道消息将会尽快发送出去（可能是马上；如果你目前离线，就会等到在线的时候）。即使关闭应用，你也知道并相信应用会在后台发送你的消息。WhatsApp 的界面甚至还以清晰简单的方式传递了这一点。如果你在离线时发送消息，它会像任何其他消息一样，进入消息流（增强你对于它不会丢失的信心），但是会用一个小手表图标来指示消息是计划发送的。一旦发送成功，手表图标就会被替换成复选标记。

采用类似的模式，我们的示例消息应用也可以使用后台同步来确保消息发送（见图 7-2）。当用户发送消息时，应用可以立即将消息添加到界面上，同时用一个小图标来展示它是计划发送的。随后，就可以使用后台同步操作将其发送到服务器，如果用户在线，就会立即完成，否则就会在用户一上线之后完成。当消息发送后，我们就可以更新界面，将计划发送的消息图标改成一个时间戳。

这种用户体验可以传达出对应用的一种信任感。通常，这种信任和技术本身一样重要。第 11 章会探讨渐进式 Web 应用中的这种和其他用户体验考量。

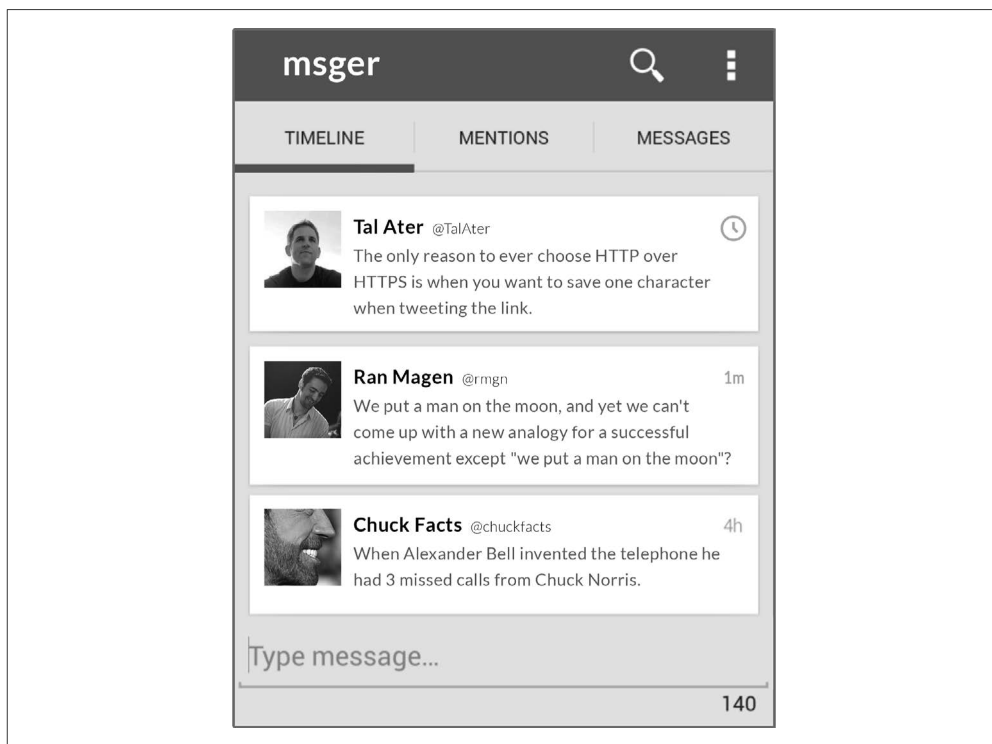


图 7-2：使用后台同步的消息应用

7.2 SyncManager

我们已经看到了注册和监听 sync 事件的代码，下面来了解其工作原理。

任何与 sync 事件的交互，都是通过 SyncManager 完成的。SyncManager 是 service worker 的一个接口，让我们可以注册 sync 事件，并获取当前已注册的 sync 事件列表。

7.2.1 访问SyncManager

我们可以通过已激活的 service worker 的 registration 对象访问 SyncManager。当你尝试从 service worker 和页面自身访问 registration 对象时，获取的方法会有所不同。

在 service worker 中，很容易通过 global 对象访问 service worker 的 registration 对象。

```
self.registration
```

在一个由 service worker 控制的页面里，可以通过调用 `navigator.serviceWorker.ready` 访问当前激活的 service worker 的 registration 对象，这个方法返回一个 promise，成功时可以拿到 service worker 的 registration 对象。

```
navigator.serviceWorker.ready.then(function(registration) {});
```

获得了 service worker 的 registration 对象后，无论是在 service worker 还是页面上，与 SyncManager 的交互操作都是相同的。

7.2.2　注册事件

要注册 sync 事件，可以在 SyncManager 中调用 register，传入一个你想要注册的 sync 事件名称（也称为“标签”）。

例如，要在 service worker 中注册一个 `send-messages` 事件，可以使用下列代码：

```
self.registration.sync.register("send-messages");
```

要在 service worker 控制的页面中注册同一个事件，可以使用下列代码：

```
navigator.serviceWorker.ready.then(function(registration) {
  registration.sync.register("send-messages");
});
```

7.2.3　sync事件

让我们回顾一下，注册 sync 事件时会发生什么。

SyncManager 维护了一个简单的 sync 事件标签列表。这个列表没有包含事件是什么或者做什么的逻辑。这些实现完全取决于 service worker 中响应 sync 事件的代码。SyncManager 只知道哪些事件被注册，何时被调用，以及如何发送 sync 事件。

当下列任何一个事件发生时，SyncManager 会给列表中的每一个注册标签发送一个 `sync` 事件：

(1) sync 事件注册后立即发送；
(2) 当用户状态从离线变成在线时；
(3) 每隔几分钟，如果有尚未完成的注册时。

在 service worker 中，发送的 sync 事件可以被监听，并使用 promise 进行响应。如果这个 promise 完成，那么对应的 sync 注册会从 SyncManager 中删除。如果 promise 拒绝，注册会保留在 SyncManager 中，并在下一个同步机会中重试。

7.2.4　事件标签

事件标签是唯一的。如果在 SyncManager 中使用一个已有的标签来注册 sync 事件，SyncManager 会忽略它，而不是添加新的条目。乍看起来这似乎有局限性，但实际上它是 SyncManager 最实用的特性之一。它允许将许多类似的操作（例如待发送的邮件）分组到单个事件中。随后，你可以注册一个 sync 事件，每次添加一个新操作到队列时，处理队列中的所有操作（例如电子邮件的发件箱），这样就不需要首先检查事件是否已经注册，或者当前是否正在运行了。

举个例子，假如你正在构建一个邮件服务。每当用户尝试发送消息时，你可以把消息保存到 IndexedDB 的发件箱中，并注册一个 `send-unsent-messages` 的后台同步事件。随后，service worker 可以包含一个事件监听器并进行响应：遍历 IndexedDB 发件箱中的每一条消

息，尝试发送它，并在成功发送后，将其从 IndexedDB 队列中删除。如果某条消息没有成功发送，整个 sync 事件就会被拒绝。SyncManager 将在稍后再次发送这个事件，允许你再次尝试清空发件箱，并确保只有在上一次事件中发送失败的消息（以及此后创建的任何新消息）才会被发送。

使用这种设置，你永远不需要检查发件箱中是否存在消息。只要有未发送的电子邮件，sync 事件就会保持注册，并定期尝试清空发件箱。在 7.4 节中，我们将在实践中看到这一点，我们在用户离线时，维护一份酒店预订的列表。

如果你认为确实需要单独的事件，可以简单地给事件提供唯一的名称，例如 `send-message-432`、`send-message-433` 等。

7.2.5 获取已注册sync事件列表

使用 SyncManager 的 `getTags()` 方法，你可以得到完整的已注册同步标签列表。

意料之中的是，和大部分 service worker 接口一样，`getTags()` 会返回一个 promise。这个 promise 完成后，会获得一个包含 sync 注册标签名称的数组。

让我们来看一个完整的例子。在 service worker 中注册一个名为 `hello-sync` 的 sync 事件，然后将当前注册的完整事件列表打印到控制台中：

```
self.registration.sync
  .register("hello-sync")
  .then(function() { return self.registration.sync.getTags(); })
  .then(function(tags) {
    console.log(tags);
  });
```

在 service worker 中运行这段代码，应该会把 `["hello-sync"]` 打印到控制台中。

在 service worker 控制的页面中，通过首先使用 `ready` 获取 registration 对象，可以取得类似的结果：

```
navigator.serviceWorker.ready.then(function(registration) {
  registration.sync
    .register("send-messages")
    .then(function() { return registration.sync.getTags(); })
    .then(function(tags) {
      console.log(tags);
    });
});
```

在 service worker 控制的页面中运行这段代码，应该会把 `["send-messages"]` 打印到控制台中。

7.2.6 最后的机会

在某些情况下，SyncManager 可能会判断出尝试发送的 `sync` 事件已经多次失败。当发生这种情况时，SyncManager 将会最后一次发送事件，给你最后一次响应它的机会。你可以通

过使用 sync 事件的 `lastChance` 属性，判断在什么时候会发生这种情况，并决定如何应对：

```
self.addEventListener("sync", event => {
  if (event.tag == "add-reservation") {
    event.waitUntil(
      addReservation()
        .then(function() {
          return Promise.resolve();
        })
        .catch(function(error) {
          if (event.lastChance) {
            return removeReservation();
          } else {
            return Promise.reject();
          }
        })
    );
  }
});
```

使用后台同步的代码出人意料地简单。然而，在现有的 Web 应用中，实现后台同步并不总是那么简单。在下一节中，我们将讨论如何解决项目中的后台同步问题。



后台同步的浏览器支持

从 Chrome 49 版本开始，可以使用后台同步。

在本书编写时，Opera、Mozilla Firefox 和 Microsoft Edge 正在实现这项功能。

7.3 传递数据给sync事件

通过把执行操作的代码从页面移动到 service worker，我们可以确保不管怎样它都将被执行，但是我们也引入了新的复杂性。

在页面中执行的大多数操作都需要依赖某些数据来完成。页面调用一个发送消息的函数时，可能需要消息的文本。一个为帖子点赞的函数可能需要帖子的 ID。但是，当我们注册 sync 事件时，唯一能传递给它的是事件名称。换句话说，你可以告诉 service worker 在后台发送消息，但是将消息文本传递给它并不像传递函数参数那样简单。

有很多方法可以解决这个问题。请允许我提出三种不同的方式。

7.3.1 在IndexedDB中维护操作队列

实现这一点的理想方法，或许是在触发后台同步操作之前，先让页面把用户操作的实体（例如消息、预订等）保存在 IndexedDB 中。随后，在 service worker 中的 sync 事件代码可以迭代对象存储，并在每个条目上执行所需的操作。一旦操作成功完成，该实体就可以从对象存储中删除。

回到我们的消息应用中，这种方法需要我们把每一条新消息添加到 `message-queue` 对象存储中，然后注册一个 `send-messages` 后台同步事件来处理它们。这个事件会遍历 `message-`

queue 中的所有消息，逐一将它们发送到网络，并最终从消息队列中删除。只有当所有消息都发送成功，对象存储为空，sync 事件才会成功完成。如果有一条消息发送失败，就会向 sync 事件返回一个拒绝的 promise，SyncManager 将会在稍后再次运行 sync 事件。

你可能希望为不同队列维护单独的对象存储（例如，一个用于消息发出，另一个用于帖子点赞），并使用不同的 sync 事件来处理它们。

使用这种方式，我们可以将以下的代码：

```
var sendMessage = function(subject, message) {
  fetch("/new-message", {
    method: "post",
    body: JSON.stringify({
      subj: subject,
      msg: message
    })
  });
};
```

替换成这样：

```
var triggerMessageQueueUpdate = function() {
  navigator.serviceWorker.ready.then(function(registration) {
    registration.sync.register("message-queue-sync");
  });
};

var sendMessage = function(subject, message) {
  addToObjectStore("message-queue", {
    subj: subject,
    msg: message
  });
  triggerMessageQueueUpdate();
};
```

然后，在 service worker 中添加下列代码：

```
self.addEventListener("sync", function(event) {
  if (event.tag === "message-queue-sync") {
    event.waitUntil(function() {
      return getAllMessages().then(function(messages) {
        return Promise.all(
          messages.map(function(message) {
            return fetch("/new-message", {
              method: "post",
              body: JSON.stringify({
                subj: subject,
                msg: message
              })
            }).then(function() {
              return deleteMessageFromQueue(message); // 返回promise
            });
          })
        );
      );
```

```
    });
  }
});
```

我们的事件监听器监听了一个名为 `message-queue-sync` 的 sync 事件，然后使用 `getAllMessages()` 获取 IndexedDB 的消息队列中的所有消息，并最终返回一个 promise 给 sync 事件，只有在其中的所有 promise 都完成时，这个 promise 才会完成。这个 promise 的创建是通过传递 promise 数组给 `Promise.all` 来完成的。我们在消息数组上运行 `map()` 方法并为每条消息返回一个 promise，以此创建了这个 promise 数组（这项技术在 4.5 节中做了解释）。只有在消息成功发送并从队列中删除之后，这些 promise 才会完成。随后，在 7.4 节中，我们将会更详细地查看一个类似的例子。

你也可以尝试一种稍微不同的方法——在一个对象存储中，同时存储队列对象和已经同步成功的对象。在使用这种技术时，你还需要保存每个对象的状态，并在对象同步成功时更新该状态。例如，你可以将应用所有的已发送和未发送消息存储在同一个对象存储中。每个消息对象除了包含消息内容，还会包含当前的状态，例如 `sent`（已发送）或者 `pending`（待发送）。随后，同步操作可以打开游标，遍历所有处于 `pending` 状态的消息，发送它们，然后将其状态改为 `sent`。在本章的后面，我们会使用这种方法来管理哥谭帝国酒店的预订。

7.3.2　在IndexedDB中维护请求队列

有时候，当你处理一个现有项目时，可能要修改应用架构来实现在本地存储对象并跟踪对象状态时，其成本很高，难以承受。有一种方法可以快速将后台同步引入到项目中，就是用请求队列替换现有的 Ajax 调用。

使用这种方式，你要将每个网络请求替换成一个将请求详情存储到 IndexedDB 的方法，随后这个方法会注册一个 sync 事件，这个事件会遍历对象存储中的所有请求，并逐个运行。

与前面的方法相反，我们的 sync 事件要在 IndexedDB 中存储复制每个网络请求所需的所有细节。同步代码不需要理解网站每个操作的意图，只需要盲目地迭代列表中的请求，并执行它们。

使用这种方式，我们可以将以下的代码：

```
var sendMessage = function(subject, message) {
  fetch("/new-message", {
    method: "post",
    body: JSON.stringify({
      subj: subject,
      msg: message
    })
  });
};

var likePost = function(postId) {
  fetch("/like-post?id="+postId);
};
```

替换成这样：

```
var triggerRequestQueueSync = function() {
  navigator.serviceWorker.ready.then(function(registration) {
    registration.sync.register("request-queue");
  });
};

var sendMessage = function(subject, message) {
  addToObjectStore("request-queue", {
    url: "/new-message",
    method: "post",
    body: JSON.stringify({
      subj: subject,
      msg: message
    })
  });
  triggerRequestQueueSync();
};

var likePost = function(postId) {
  addToObjectStore("request-queue", {
    url: "/like-post?id="+postId,
    method: "get"
  });
  triggerRequestQueueSync();
};
```

我们将所有的网络请求替换为这样的代码：将代表请求的对象存储到名为 `request-queue` 的对象存储中。这个存储中的每个对象都代表着一个网络请求，其中包含了需要复制的每一条信息。接下来，我们可以添加一个 sync 事件监听器到 service worker 中，它负责遍历 `request-queue` 的所有请求，逐一发起网络请求，然后从对象存储中将其删除：

```
self.addEventListener("sync", function(event) {
  if (event.tag === "request-queue") {
    event.waitUntil(function() {
      return getAllObjectsFrom("request-queue").then(function(requests) {
        return Promise.all(
          requests.map(function(req) {
            return fetch(req.url, {
              method: req.method,
              body: req.body
            }).then(function() {
              return deleteRequestFromQueue(message); // 返回一个promise
            });
          })
        );
      });
    });
  }
});
```

已完成的请求会从 IndexedDB 队列中删除（使用 `deleteRequestFromQueue()`）。失败的请求会保留在队列中，并返回拒绝的 promise。如果有一个或者多个请求返回了失败的 promise，那么请求队列将会在下一次 sync 事件中再次迭代（这一次不会再包含已经成功

发起的请求)。

要了解从对象存储中获取对象的示例函数实现，以及其他的 IndexedDB 代码，请参见第 6 章。

7.3.3 传递数据给sync事件标签

当你只需要传递一个简单的值给 sync 函数时，实现一个数据库来跟踪每一个操作，有时候看起来是杀鸡用牛刀。接下来介绍的技巧肯定看起来有点不完善，但是有时候你要寻找的，正是一种快速上手的解决方案。

假设你的页面允许用户“点赞”某个帖子，要进行一个只需要把帖子 ID 发送到某个 URL 的动作。你的现有代码可能是这样：

```
var likePost = function(postId) {
  fetch("/like-post?id="+postId);
};
```

正如我们以前看到的那样，你可以将这段代码替换成一个 IndexedDB 队列，其中包含了需要点赞的帖子，然后遍历这些帖子。但有时候，让事情保持简单是有价值的。用下列代码来替换 `likePost` 函数可以取得类似的效果，而不需要维护一个帖子的数据库：

```
var likePost = function(postId) {
  navigator.serviceWorker.ready.then(function(registration) {
    registration.sync.register("like-post-"+postId);
  });
};
```

我们的 sync 事件代码也能做到如此简单，简单地判断事件名称是否以 `like-post-` 开头，然后从中提取帖子的 ID：

```
self.addEventListener("sync", function(event) {
  if (event.tag.startsWith("like-post-")) {
    event.waitUntil(function() {
      var postId = event.tag.slice(10);
      return fetch("/like-post?id="+postId);
    });
  }
});
```

7.4 给应用添加后台同步

现在，我们已经对后台同步有了基本的了解，是时候动手用它来改进哥谭帝国酒店的 Web 应用了。

My Account 页面顶部是一个可以让用户发起新预订的表单。当用户提交这个表单时，`my-account.js` 中的 `addReservation()` 函数会被调用。这个函数会从表单输入中创建一个新的 `reservationDetails` 对象，并给它设置一个 `Awaiting confirmation`（待确认）状态。随后，将这个对象添加到 IndexedDB 的 `reservations` 对象存储中，渲染到 DOM 中，并最终向服务器发起 Ajax 请求，向酒店发起预订。

但是，假设网络永远可用就是自找麻烦。我们在本地机器上测试时可能一切正常，但是如果用户试图在丢失连接的情况下发起预订，那么这段逻辑就会失败。如果在用户离线时调用了 `addReservation()`，那么新的预订会写入 IndexedDB 并渲染到页面，但是 Ajax 请求会失败，服务器不知道发起了新的请求。预订会出现在页面上，并保存在 IndexedDB 中，甚至在用户刷新浏览器后也会保留在那里。无论用户如何操作，他都会看到预订无限期地处于 `Awaiting confirmation` 状态，而且服务器是完全不知情的。用户会因此感到无比沮丧，也严重损害了酒店股东的利益。

我们可以通过将创建新预订的请求，从页面搬到 service worker 中的 sync 事件，来解决这个问题。

以下是我们需要完成的步骤。

(1) 修改 `addReservation()` 函数，检查浏览器是否支持后台同步。如果支持，则注册一个 `sync-reservations` 同步事件。否则就和之前一样，使用常规的 Ajax 调用。
(2) 添加新预订到 IndexedDB 中的代码，需要把新预订的状态改为 `Sending`（发送中）。在预订成功添加到服务器之前，这就是用户看到的状态，添加成功后，服务器会返回新的状态（`Awaiting confirmation` 或者是 `Confirmed`）。
(3) 我们会向 service worker 添加一个事件监听器，用来响应 sync 事件。如果检测到的 sync 事件名称是 `sync-reservations`，事件监听器就会遍历每一个处于 `Sending` 状态的预订，并尝试将其发送到服务器。成功添加到服务器之后，IndexedDB 中的预订会被修改为新的状态。如果任何服务器请求失败，整个 sync 事件就会被拒绝，浏览器会尝试在随后再次运行这个事件。

首先，我们要修改 `addReservation()`，用来检查后台同步是否在当前浏览器中可用。如果是的话，就会注册一个 sync 事件，而不是直接调用服务器。

但是在开始之前，要在命令行中运行下列命令，确保代码处于上一章结束时的状态：

```
git reset --hard
git checkout ch07-start
```

接下来，在 my-account.js 中，按照以下示例修改 `addReservation()` 的代码：

```
var addReservation = function(id, arrivalDate, nights, guests) {
  var reservationDetails = {
    id:          id,
    arrivalDate: arrivalDate,
    nights:      nights,
    guests:      guests,
    status:      "Sending"
  };
  addToObjectStore("reservations", reservationDetails);
  renderReservation(reservationDetails);
  if ("serviceWorker" in navigator && "SyncManager" in window) {
    navigator.serviceWorker.ready.then(function(registration) {
      registration.sync.register("sync-reservations");
    });
  } else {
    $.getJSON("/make-reservation", reservationDetails, function(data) {
```

```
      updateReservationDisplay(data);
    });
  }
};
```

我们首先把 addReservation() 函数创建 reservationDetails 对象的状态从 Awaiting confirmation 改成了 Sending。随后，检查当前浏览器是否同时支持 ServiceWorker 和 SyncManager。如果支持，我们就注册一个 sync-reservations 同步事件。否则，就和以前一样，使用 $.getJSON 在页面上进行预订。

在为这个事件创建事件监听器之前，我们先对 reservations-store.js 文件做出两处小改进。这些改进可以让我们轻松获取到 Sending 状态的预订。

首先，我们在 reservations 存储中，给 status 字段添加一个新的索引。

在 reservations-store.js 中，将第一行的 DB_VERSION 从 1 改为 2（如果你创建了更多版本，则改成更高的版本）。接下来，在同一个文件里，修改 openDatabase() 函数中的 onupgradeneeded 函数，为 reservations 对象存储中的 status 字段创建索引。代码如下所示：

```
request.onupgradeneeded = function(event) {
  var db = event.target.result;
  var upgradeTransaction = event.target.transaction;
  var reservationsStore;
  if (!db.objectStoreNames.contains("reservations")) {
    reservationsStore = db.createObjectStore("reservations",
      { keyPath: "id" }
    );
  } else {
   reservationsStore = upgradeTransaction.objectStore("reservations");
  }

  if (!reservationsStore.indexNames.contains("idx_status")) {
    reservationsStore.createIndex("idx_status", "status", { unique: false });
  }
};
```

这段代码中演示了一些我们还没接触过的内容。在第 6 章中，我们只看到了如何在新的对象存储上创建索引。这一次，由于对象存储中可能已经存在了一些用户数据，我们需要将索引添加到现有的对象存储中，或者是创建新的对象存储并添加索引。

我们的代码仍然遵循了 6.2.5 节中的版本管理模式，它主张在每次进行修改之前，先确认修改是否有必要。在创建 reservations 对象存储之前，我们先判断它是否存在。如果不存在，我们就进行创建，并将引用保存到 reservationsStore 变量中。如果已经存在，我们就通过调用 event.target.transaction.objectStore("reservations") 获得更新事件中的事务，并从事务中获取 reservations 对象存储的引用。

最后，当我们确认 reservations 对象存储已经存在时（要么在前一个版本中已经创建，要么是因为刚才已经创建），就可以检查对象存储的 indexNames 属性，判断其是否已经包含了我们需要的索引。如果没有包含，我们就继续创建它。

在 reservations-store.js 中的最后一处修改，让我们可以使用这个新的索引，轻松获取到处

于某个状态的所有预订。要做到这一点，我们要改进 `getReservations` 函数，让其支持接收两个可选的参数：索引名称，以及传递给该索引的值。

在 reservations-store.js 中修改 `getReservations()` 函数，如下所示：

```
var getReservations = function(indexName, indexValue) {
  return new Promise(function(resolve) {
    openDatabase().then(function(db) {
      var objectStore = openObjectStore(db, "reservations");
      var reservations = [];
      var cursor;
      if (indexName && indexValue) {
        cursor = objectStore.index(indexName).openCursor(indexValue);
      } else {
        cursor = objectStore.openCursor();
      }
      cursor.onsuccess = function(event) {
        var cursor = event.target.result;
        if (cursor) {
          reservations.push(cursor.value);
          cursor.continue();
        } else {
          if (reservations.length > 0) {
            resolve(reservations);
          } else {
            getReservationsFromServer().then(function(reservations) {
              openDatabase().then(function(db) {
                var objectStore =
                  openObjectStore(db, "reservations", "readwrite");
                for (var i = 0; i < reservations.length; i++) {
                  objectStore.add(reservations[i]);
                }
                resolve(reservations);
              });
            });
          }
        }
      };
    }).catch(function() {
      getReservationsFromServer().then(function(reservations) {
        resolve(reservations);
      });
    });
  });
};
```

新的函数包含了两处修改。首先，它允许 `getReservations()` 接收两个可选参数（`indexName` 和 `indexValue`）。其次，如果函数接收了这些参数，就使用参数在特定索引（`indexName`）上打开游标，而不是直接打开对象存储。随后打开特定值（`indexValue`）的游标，会把结果限制指定的范围内。如果没有传递这些参数，它将会像以前那样运行，并返回所有的预订。

做出这两处修改后，我们的函数可以返回所有结果，也可以仅返回结果的一个子集，如下所示：

```
getReservations().then(function(reservations) {
  // reservations变量包含了所有预订
});

getReservations("idx_status", "Sending").then(function(reservations) {
  // reservations变量仅包含了状态为"Sending"的预订
});
```

现在，一切已经准备好，可以在 service worker 中处理未发送的预订了。我们可以继续添加后台同步的事件监听器到 service worker 中。

首先，要确保在 serviceworker.js 的第一行引入 reservations-store.js 文件。代码应该如下所示：

```
importScripts("/js/reservations-store.js");
```

接下来，在 serviceworker.js 的底部，添加下列代码：

```
var createReservationUrl = function(reservationDetails) {
  var reservationUrl = new URL("http://localhost:8443/make-reservation");
  Object.keys(reservationDetails).forEach(function(key) {
    reservationUrl.searchParams.append(key, reservationDetails[key]);
  });
  return reservationUrl;
};

var syncReservations = function() {
  return getReservations("idx_status", "Sending").then(function(reservations) {
    return Promise.all(
      reservations.map(function(reservation) {
        var reservationUrl = createReservationUrl(reservation);
        return fetch(reservationUrl);
      })
    );
  });
};

self.addEventListener("sync", function(event) {
  if (event.tag === "sync-reservations") {
    event.waitUntil(syncReservations());
  }
});
```

在深入研究 `createReservationUrl()` 和 `syncReservations()` 的细节之前，我们先来看看这段新代码的最后一部分。我们使用 `self.addEventListener` 为 sync 事件添加了一个新的事件监听器。这个事件监听器会响应标签为 `sync-reservations` 的事件，让其 `waitUntil` 等待 `syncReservations()` 返回的 promise，根据这个 promise 完成或者拒绝来判断 sync 事件是完成还是拒绝。如果 `syncReservations()` 返回的 promise 完成，那么 `sync-reservations` sync 事件就会从 SyncManager 中删除（直到我们再一次注册它）。如果 promise 拒绝，那么 SyncManager 会保持 sync 事件的注册，并在随后再次触发该事件。

通过 `syncReservations()` 创建的能够决定整个 sync 事件结果的 promise 是什么？广义地说，`syncReservations()` 会遍历 IndexedDB 中每一个被标记为 `Sending` 状态的预订，尝试将其发送到服务器，并返回一个 promise，只有当每一个预订都发送成功时，这个 promise

才会解决。如果一个预订失败了，那么 syncReservations() 返回的整个 promise 就会失败。

为了实现这一点，syncReservations() 首先使用 getReservations() 函数，获取了所有处于 Sending 状态的预订。getReservations() 函数返回了一个 promise，其中包含了所有需要发送的预订。随后，我们使用 Promise.all() 将所有单独的 promise 包裹起来，并返回一个单独的 promise，这个 promise 会决定整个 syncReservations() 函数的结果。

要做到这一点，我们需要给 Promise.all() 传入一个 promise 数组。我们拿到预订对象的数组后，通过使用 Array.map() 方法将数组元素转换为 promise，从而创建出这个数组。我们使用 map() 对每个预订进行迭代，创建一个 fetch 请求发送到服务器来创建这个预订。fetch() 会返回一个 promise，这个 promise 正是我们要返回并放入 promise 数组中，并传递给 Promise.all() 的。

关于如何使用 Promise.all() 和 Array.map() 创建一个 promise 数组，请参见 4.5 节中的“为 Promise.all() 创建一个 promise 数组”。

最后我们来看 createReservationUrl() 函数。这个函数使用 URL 接口创建了一个新的 URL 对象，这个对象表示了 fetch 请求要发往的 Web 地址。这个代码仅仅是用一种更优雅的方式来创建带有查询字符串的 URL，而不是手工拼接字符串、值、& 符号和问号。这个函数接收的对象包括了预订的详情，并返回一个 URL 对象，其中包含了查询字符串的详情：

```
console.log(
  createReservationUrl({nights: 2, guests: 4});
);
// 返回一个新的URL对象，指向的地址是
// http://localhost:8443/make-reservation?nights=2&guests=4
```

完成上述所有修改后，可以再次访问 My Account 页面。这一次，一旦页面加载，模拟离线状态（使用浏览器的开发者工具，或者停止开发服务器），并尝试发起新的预订。预订会被添加到 IndexedDB 和 DOM 中，sync 事件会注册，但是服务器是无法到达的。预订依然会停留在 Sending 状态，如图 7-3 所示。接下来，将服务器的连接恢复，在几分钟内，sync 事件会重新发送，预订会被修改为 Confirmed 状态。

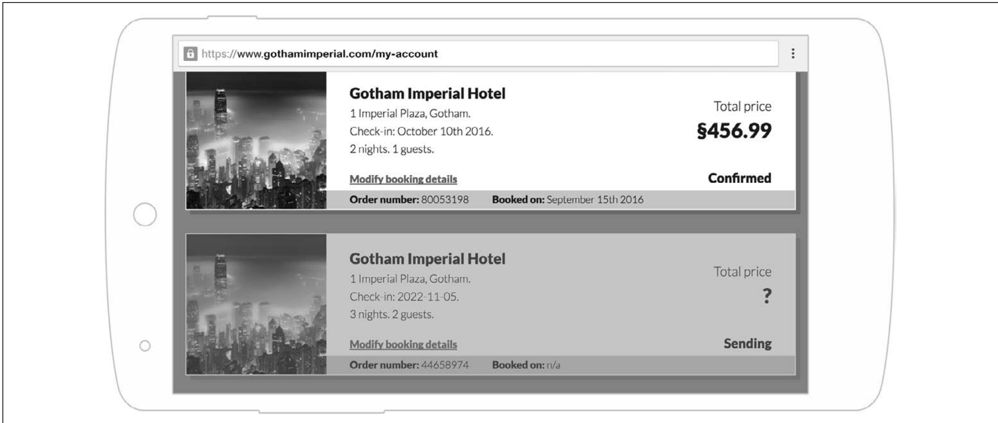


图 7-3：使用后台同步发起预订



如果你恢复连接后，一直在等待 sync 事件运行，并开始怀疑它是否已经运行，可以在浏览器控制台中运行以下代码：

```
navigator.serviceWorker.ready.then(function(registration) {
  registration.sync.getTags().then(function(tags) {
    console.log(tags);
  });
});
```

这段代码会输出当前已注册 sync 事件的完整列表。如果预订 sync 事件依然在注册，那么代码应该会把 ["sync-reservations"] 打印到控制台中。

后台同步最令人印象深刻的是，即使关闭哥谭帝国酒店网站后，sync 事件也依然会到达服务器。SyncManager 会跟踪所有等待中的 sync 注册，并不知疲倦地确保事情可以在后台完成。如果没有 service worker——一个即使在用户关闭你的渐进式 Web 应用之后还能进行响应的脚本，这是不可能实现的。

这引出了 sync 事件中缺少的最后一个步骤。当 sync 事件成功创建预订之后，fetch 请求会返回一个新的预订详情对象，其中包含了新的细节，包括预订的最终价格，以及更新后的预订状态。我们需要更新 IndexedDB 中的预订详情，以便显示最新的信息给用户。更重要的是，我们需要更新预订状态，以便在下一次 sync-reservations 事件注册的时候，预订不会被重复发送。

在 serviceworker.js 中更新 syncReservations() 函数，如下所示：

```
var syncReservations = function() {
  return getReservations("idx_status", "Sending").then(function(reservations) {
    return Promise.all(
      reservations.map(function(reservation) {
        var reservationUrl = createReservationUrl(reservation);
        return fetch(reservationUrl).then(function(response) {
          return response.json();
        }).then(function(newReservation) {
          return updateInObjectStore(
            "reservations",
            newReservation.id,
            newReservation
          );
        });
      })
    );
  });
};
```

syncReservations() 最新版本的唯一变化，是当 fetch() 完成时，我们不再立即认为 promise 完成。现在，当 fetch 返回的 promise 完成时，then 中的新函数会被调用，其中包含了 fetch 的响应。这个对象中包含了 JSON，是我们通过调用 response.json() 解析得到的。这会返回一个 promise，其中包含了预订详情的简单 JavaScript 对象，我们在 then 中，将这个对象传递给 updateInObjectStore() 函数。

现在，即使在离线时进行预订的用户，也会在他们的本地 IndexedDB 对象存储中获得最新预订数据。即使 sync 事件在用户离开网站之后成功发起预订，我们也能确保 IndexedDB 对象存储中的数据能够保持最新。

7.5 小结

在现代渐进式 Web 应用中，后台同步有潜力成为其中一个最重要的组成部分。这是其中一项对用户体验至关重要的技术，但是它对用户是不可见的——直到它停止工作为止。

当你着手将后台同步添加到自己的应用中时，可能会遇到两项主要的挑战。

首先要将逻辑（连同运行所需的所有数据）从页面移动到 service worker 中。这是本章所解决的问题。

第二个问题是将后台同步事件的结果传递回页面和用户。你经常需要基于后台同步操作的结果来修改页面（例如，可视化地将消息标记为“已发送”，或者将帖子标记为“已点赞”）。由于 service worker 不能直接访问页面窗口，所以我们需要一种方法来将这些操作的结果从 service worker 返回到页面。在第 8 章中，我们将探讨如何通过在 service worker 和页面之间传递消息来实现这一点。

但是这又引出了另一项有意思的挑战。如果 sync 事件成功发生时，用户已经离开了站点呢？我们如何让用户知道预订已经接收？如何在稍后状态改变时通知用户？在第 10 章中，我们将学习如何使用推送通知来始终让用户了解最新状态。

第 8 章

使用postMessage()在service worker和页面之间通信

当我们把越来越多的逻辑从页面转移到 service worker 之后，经常会发现需要在两者之间进行通信。

在第 7 章中我们了解到，将诸如网络请求之类的重要事件从不稳定的页面移动到 service worker 中，可使应用更加可靠。但是，我们经常需要根据这些操作的结果更新页面。例如，在 7.4 节中，我们将发起新预订的代码移动到了运行在 service worker 中的后台同步事件中。这个事件调用了服务器，并接收了一个 JSON 文件作为响应，其中包含了更新后的预订详情。我们使用 JSON 文件中的数据，更新了 IndexedDB 中的预订详情，但是由于 service worker 不能访问窗口，我们不能将预订详情更新到 DOM 中。取而代之的是，页面依赖于一个朴素的 `setInterval()` 方法，每隔几秒通过网络检查一下预订状态，并更新 DOM。如果 sync 事件可以在接收到更新过的预订详情之后立刻将其发送到页面中，我们就可以立刻更新 DOM，而不需要发起不必要的网络请求了。

本章，我们将看看如何使用 `postMessage()` 在页面和 service worker 之间来回发送消息和数据，并探索几种类型的通信：

- 从窗口向控制它的 service worker 发送消息
- 从 service worker 向作用域内的所有窗口发送消息
- 从 service worker 向特定窗口发送消息
- 通过 service worker 在窗口之间发送消息

8.1　窗口向service worker通信

从页面向 service worker 发送消息很简单。

从页面发送消息之前，首先要获取当前控制页面的 service worker。可以使用 `navigator.service Worker.controller` 获取这个 service worker。

接下来，可以使用 service worker 的 `postMessage()` 方法，该方法接收的第一个参数就是消息本身。消息可以是几乎任何值，或者 JavaScript 对象，包括字符串、对象、数组、数字、布尔类型等。

下面的示例展示了从页面向 service worker 发送一条包含了一个简单对象的消息：

```
navigator.serviceWorker.controller.postMessage(
  {arrival: "05/11/2022", nights: 3, guests: 2}
)
```

消息一旦发布，service worker 就可以通过监听 `message` 事件来捕获它：

```
self.addEventListener("message", function (event) {
  console.log(event.data);
});
```

本例中的代码会监听传入的消息，并将消息内容记录到控制台中。消息内容可以在传递给事件监听器（`event.data`）的 `event` 对象的 `data` 属性中找到。

除了包含消息数据本身之外，`event` 对象还包含了其他许多有用的属性。其中一些最实用的属性在 `source` 属性中。`source` 包含了发送消息的窗口的相关信息，可以帮助我们决定该做什么，以及向哪里发送消息响应。以下是 message 事件 `source` 属性的一些示例用法：

```
self.addEventListener("message", function (event) {
  console.log("Message received:", event.data);
  console.log("From a window with the id:", event.source.id);
  console.log("which is currently pointing at:", event.source.url);
  console.log("and is", event.source.focused ? "focused" : "not focused");
  console.log("and", event.source.visibilityState);
});
```

让我们看一个向 service worker 发送消息的可能用例。

哥谭帝国酒店可能会决定对其 Web 应用进行扩展，增加旅游指南，列出哥谭的每一家餐馆。由于哥谭有数以千计的餐馆，我们可能觉得缓存每一家餐馆的详情会太多了。我们可以选择只缓存用户查看过的餐馆的详情。

要实现这一点，我们可以添加代码，从餐馆详情页面发送消息：

```
navigator.serviceWorker.controller.postMessage("cache-current-page");
```

当用户访问一家餐馆的页面时，会有一条消息发送到 service worker。service worker 可以监听这些消息，并使用事件的 `source` 属性，判断需要缓存哪个页面：

```
self.addEventListener("message", function (event) {
  if (event.data === "cache-current-page") {
```

```
    var sourceUrl = event.source.url;
    if (event.source.visibilityState === "visible") {
      // 立即缓存sourceUrl和相关文件
    } else {
      // 将sourceUrl和相关文件添加到队列中，稍后缓存
    }
  }
});
```

这个示例中的代码使用消息的源 URL 来确定需要缓存哪个页面。它还根据页面的当前可见状态来判断应首先请求并缓存哪个页面。这样，如果用户在许多单独的标签页中打开了一堆餐馆，那么当前可见标签中的内容就会被先缓存。在 11.4 节中我们将会看到如何在页面被缓存之后更新页面及其 UI，让用户知道现在页面已经缓存并且离线可用。



请注意，当前页面需要有一个控制它的 service worker，否则调用 `navigator.service Worker.controller.postMessage()` 会导致报错。如果用户第一次访问网站，可能会安装并激活新的 service worker，但这并不意味着它正在控制当前页面。在这种情况下，`navigator.serviceWorker.controller` 会是 undefined，然后代码会中断，因为 undefined 没有包含 `postMessage()` 方法。在第 4 章中，你可以读到更多关于 service worker 从安装到激活并控制页面的相关信息。

实际上，应该重写上述代码，在尝试使用 service worker 之前，引入判断 service worker 是否存在的检查：

```
if ("serviceWorker" in navigator
    && navigator.serviceWorker.controller) {
  navigator.serviceWorker.controller.postMessage(
    "cache-current-page"
  );
}
```

8.2 service worker向所有打开的窗口通信

从 service worker 向页面发送消息，类似于从页面向 service worker 发送消息，唯一的区别是在哪个对象上调用 `postMessage()`。目前为止，我们只在 service worker 上调用过 `postMessage()`，这次我们将在 service worker 的客户端上调用它。

在 service worker 内，我们可以使用 service worker 的全局对象中的 `clients` 对象，获取 service worker 作用域内所有当前打开的窗口（WindowClient）。`clients` 包含了一个 `matchAll()` 方法，我们可以用这个方法获取 service worker 作用域内所有当前打开的窗口（客户端）。`matchAll()` 返回一个 promise，在完成时，返回一个包含 0 个或者多个 WindowClient 对象的数组：

```
self.clients.matchAll().then(function(clients) {
  clients.forEach(function(client) {
    if (client.url.includes("/my-account")) {
      client.postMessage("Hi client: "+client.id);
    }
  });
});
```

这段代码获取了这个 service worker 当前控制的所有客户端，对它们进行迭代，并向当前显示 My Account 页面的客户端发送消息。

在页面中监听来自 service worker 的 message 事件，和我们在 8.1 节中看到的非常类似。但是这次，我们要将事件监听器添加到 `serviceWorker` 对象中：

```
navigator.serviceWorker.addEventListener("message", function (event) {
  console.log(event.data);
});
```

如果在页面中包含了这段代码，并在 service worker 中运行了之前的代码，当前指向 My Account 页面的任何页面都会把消息打印到控制台中，消息内容类似于：

```
Hi client: b85b7e3d-a893-4b67-9e41-1d6fddf40110
```



仅仅把代码放在 service worker 的顶部是不够的。如果代码放置在事件之外，它只会在 service worker 脚本加载后、service worker 安装前以及任何客户端监听之前，执行一次。相反，要将其添加到事件中，如下面的代码示例所示。在开发期间，还可以使用浏览器控制台，在 service worker 的作用域内运行这段代码。具体请参见 4.8.1 节。

让我们来看看这类通信的典型用例。

我们想要向哥谭帝国酒店应用的用户保证，无论他们在线还是离线，都可以使用这款应用。为此，我们可以在 service worker 安装并缓存所需的一切静态资源之后，立即向用户显示一条消息。下面的代码示例修改了 install 事件，在缓存完成之后向所有客户端发送消息。然后，页面可以响应这个事件，向用户显示一条消息，向他们保证这款应用在离线和在线状态下都能使用。

```
self.addEventListener("install", function(event) {
  event.waitUntil(
    caches.open(CACHE_NAME).then(function(cache) {
      return cache.addAll(CACHED_URLS);
    }).then(function() {
      return self.clients.matchAll({ includeUncontrolled: true });
    }).then(function(clients) {
      clients.forEach(function(client) {
        client.postMessage("caching-complete");
      });
    })
  );
});
```

这段代码和哥谭帝国酒店现有的 install 事件状态很类似，只添加了一处。一旦 `cache.addAll()` 返回的 promise 成功，我们就使用 `clients` 对象，获取当前打开的所有 WindowClient 对象，并向每个客户端发送一条消息。

发送消息的代码基于我们在本节第一个示例中看到的相同原则，但是我们引入了一处重要的变化。当调用 `clients.matchAll()` 时，我们传入了一个选项对象，让其包含未受控制的

客户端。对于开发者来说，这个例子也体现了理解 service worker 生命周期的重要性（正如第 4 章中解释的那样）。当用户第一次访问页面时，service worker 会安装并激活。然而，页面此时依然不受 service worker 控制。如果我们没有让 `self.clients.matchAll()` 包括未受控制的窗口，我们的消息就不会到达目的地。

在第 11 章中，我们将会介绍一个在缓存完成时通知用户的完整例子。

8.3　service worker向特定窗口通信

除了 `matchAll()` 方法之外，`clients` 对象还有另一个实用的方法，让你可以通过 `get()` 获取单个客户端对象。通过传递一个已知客户端的 ID 给 `get()`，我们就可以得到一个 promise，当其完成时会得到 WindowClient 对象。之后我们就可以使用这个对象，给该客户端发送消息。

举个例子，如果我们知道其中一个 WindowClient 的 ID 是 `d2069ced-8f96-4d28`，就可以运行下面的代码，让窗口知道它当前是否可见：

```
self.clients.get("d2069ced-8f96-4d28").then(function(client) {
  client.postMessage("Hi window, you are currently " + client.visibilityState);
});
```

有几种方式可以找到客户端窗口的 ID。一种方式是在使用 `clients.matchAll()` 迭代所有打开的客户端时，通过 WindowClient 对象的 `id` 属性获取。另一种可能的方法是，通过 post message 事件的 `source` 属性获取。这两种方式的用法如下所示：

```
self.clients.matchAll().then(function(clients) {
  clients.forEach(function(client) {
    self.clients.get(client.id).then(function(client) {
      client.postMessage("Messaging using clients.matchAll()");
    });
  });
});

self.addEventListener("message", function(event) {
  self.clients.get(event.source.id).then(function(client) {
    client.postMessage("Messaging using clients.get(event.source.id)");
  });
});
```

没错，这两个示例用法都是非常多余的。在这两个例子中，我们使用客户端对象（在第一个例子中是 `client`，第二个例子中是 `event.source`）来获取其 ID，然后又通过 ID 获取对应的客户端对象。这两个例子都可以进行简化，以避免使用 `clients.get()`：

```
self.clients.matchAll().then(function(clients) {
  clients.forEach(function(client) {
    client.postMessage("Messaging using clients.matchAll()");
  });
});

self.addEventListener("message", function (event) {
```

```
  event.source.postMessage("Messaging using event.source");
});
```

一种更有可能使用 clients.get() 的场景，是把客户端 ID 存储在 service worker 中，以便随后使用 clients.get() 进行访问。

举个例子，设想一个应用是用来跟踪股票市场的。这个应用使用一个数据流，许多不同股票的更新都是通过这个数据流到达的。当知道用户倾向于打开多个窗口，每个窗口显示在不同的屏幕上并且跟踪不同的股票时，你会意识到你不得不在所有打开的窗口中保持同一个数据流打开。在试图优化应用以节省带宽和服务器成本时，你决定停止在每个单独的窗口中打开流，而是在 service worker 中打开一个流，处理所有的股票价格变化。随后，每个页面在打开时，可以向 service worker 发送消息，告诉 service worker 它想要订阅哪支股票的更新。service worker 会维护一份列表，其中记录了每个客户端 ID 想要更新哪支股票。现在，任何时候，只要关于特定股票的更新通过流到达，service worker 都可以获取一份对该股票感兴趣的客户端列表，并使用 client.get() 给每个客户端发送更新后的股票信息。

8.4　使用MessageChannel保持通信渠道打开

目前为止，我们只看到了如果使用 WindowClient 或者 service worker 对象发送消息，并且只看到了 postMessage() 接收的第一个参数。但实际上，postMessage() 可以接收第二个参数，你可以使用这个参数来保持双方之间的通信渠道打开，来回发送消息。

这种通信是通过 MessageChannel 对象处理的。

如果你熟悉这个实验：用一条线把两个杯子连接起来，一个人对着杯子说话，另一个人通过杯子听，那么你就已经熟悉 MessageChannel 的工作原理了。

在 MessageChannel 中，两个杯子被称为 port1 和 port2（见图 8-1）。你可以通过 postMessage() 对着每个杯子（或者端口）说话，并且可以使用事件监听器监听每个杯子。

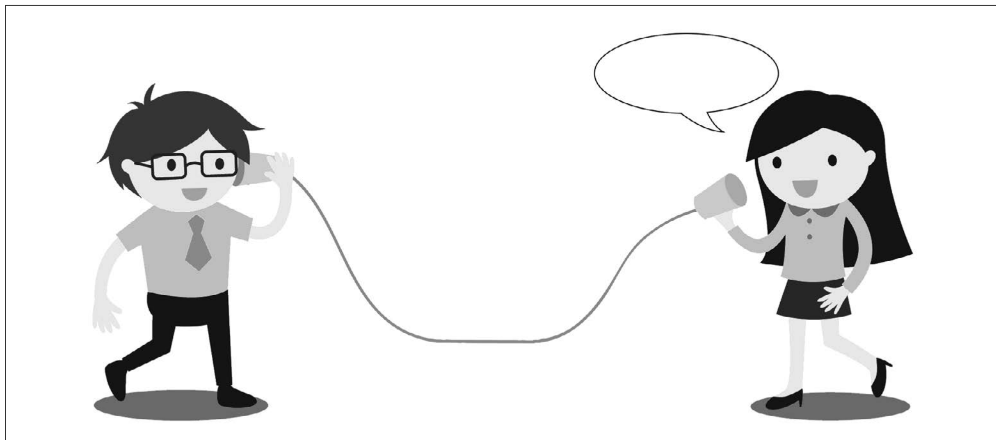


图 8-1：Throw new ClipartException('string not pulled taut');

```
var msgChan = new MessageChannel(); msgChan.port1.onmessage = function(msg) {
  console.log("Message received at port 1:", msg.data);
};
msgChan.port2.postMessage("Hi from port 2");
```

这段代码创建了一个新的“杯子电话”MessageChannel，监听杯子 `port1`，并且对着另一个杯子（`port2`）说话。在浏览器中运行这段代码，应该能够把 `Message received at port 1: Hi from port 2` 打印到控制台中。

当我们从窗口向 service worker 通信时（反之亦然），可以在窗口中创建一个新的 MessageChannel 对象，并通过 `postMessage` 将其中的一个端口传递给 service worker。当消息到达后，就可以在 service worker 中访问端口了。结果是我们在 service worker 和窗口之间打开了一条通信渠道，两者各拥有一个端口。

```
// 窗口代码
var msgChan = new MessageChannel();
msgChan.port1.onmessage = function(event) {
  console.log("Message received in page:", event.data);
};

var msg = {action: "triple", value: 2}; navigator.serviceWorker.controller.
postMessage(msg, [msgChan.port2]);


// service worker代码
self.addEventListener("message", function (event) {
  var data = event.data;
  var openPort = event.ports[0];
  if (data.action === "triple") {
    openPort.postMessage(data.value*3);
  }
});
```

页面上的代码首先创建了一个新的 MessageChannel，并在第一个端口上添加了事件监听器，以将收到的任何信息打印出来。接下来，代码向 service worker 发送了一条消息，同时将 MessageChannel 的第二个端口传递过去。请注意，`postMessage` 接收一个端口数组作为其第二个参数，以便你可以通过 0 个或者多个端口进行通信。

同时，在 service worker 中，我们监听了发送到 service worker 的 `message` 事件。当检测到这样的事件时，其中的 `event` 对象同时会包含消息内容（`event.data`）和页面发送的端口数组（`event.ports`）。我们的事件监听器会检查消息，如果 `data` 对象中包含了 `action` 属性，并且值为 `triple`，那么就会将 `data` 对象中的 `value` 属性乘以 3，然后将消息发送回去。这个消息是通过在 `event.ports[0]` 中找到的 MessageChannel 端口直接发送的，也就是页面中创建的 MessageChannel 的 `port2`。随后，消息会沿着从 service worker 到页面的“绳子”到达 `port1`，在那里有一个单独的事件监听器会将其打印到控制台中。

这个简单的例子展示了如果将数学计算从页面委派给 service worker。类似地，页面可以询问 service worker 缓存中是否存在某一项，或者当前打开了多少个展示应用的标签。service worker 也可以通过反向询问，向其控制的窗口询问输入字段的值，甚至是用户在页面中的滚动距离，以便开始缓存下一页。



我鼓励你再次看看之前的例子，对比调用 `postMessage()` 的不同方式，以及添加事件监听器的不同方式。请注意，我们可以在 service worker 对象和 MessageChannel 端口上调用 `postMessage()`。类似地，有时候我们在 service worker 上监听消息事件，有时候则是在 MessageChannel 端口上监听。

如果事情没有像预料的那样发生，请检查你是否将事件绑定在了正确的对象上，以及是否将消息发送给了正确的对象。如果 service worker 将消息发送到一个 MessageChannel 端口，而页面在 service worker 上而不是另一个端口上监听了 `message` 事件，就什么也不会发生——你把耳朵放在了盘子上，而不是放在另一个杯子上。

前面的示例演示了如何使用 MessageChannel 来响应 `postMessage`。让我们来看看另一个例子，它展示了如何在页面和 service worker 之间保持持续的通信通道打开。

```
// 窗口代码
var msgChan = new MessageChannel(); msgChan.port1.onmessage = function(event) {
  console.log("URL fetched:", event.data);
};
navigator.serviceWorker.controller.postMessage("listening", [msgChan.port2]);


// service worker代码
self.addEventListener("message", function (messageEvent) {
    var openPort = messageEvent.ports[0];
    self.addEventListener("fetch", function(fetchEvent) {
      openPort.postMessage(fetchEvent.request.url);
    });
});
```

这个例子中的窗口代码和前一个例子非常相似。我们创建了一个新的 MessageChannel，监听一个端口，并发送消息给 service worker，其中包含另一个端口。唯一的变化是消息的内容。

当 service worker 接收到这条消息后，它为自己的 `fetch` 事件添加了一个事件监听器，让其通过 `openPort` 发送消息，其中包含了每次 `fetch` 请求的 URL。

结果是页面会持续记录每个网络请求的 URL——不仅包含当前标签页的请求，还包含了这个 service worker 控制的其他窗口的请求。将这个文件命名为 network.html，并在另一个选项卡中浏览这个页面，此时你已经完成了构建属于自己的浏览器开发者工具的第一步了。

8.5 窗口间的通信

让我们结合目前所学的内容，看看如何在不同窗口间进行通信。过去，在不同窗口间传递消息需要借助于一些技巧，例如在 cookie、localStorage 甚至服务端写入消息。但是，service worker 提供了一个中心连接点，可触达作用域内每一个打开的窗口，使我们最终可以在窗口间发送消息、对象，甚至是 MessageChannel 端口。

是时候回到实际编码中了。

在开始前，可以通过在命令行中运行以下命令，确保代码处于上一章结束时的状态：

```
git reset --hard
git checkout ch08-start
```

在哥谭帝国酒店 My Account 页面的顶部，有一个登出账号的链接。在点击时，该链接会把用户送回网站首页（这个应用建立在信任的基础上，不需要输入用户名或者密码。在你的应用中，可以根据实际情况在这里包含一些登录 / 注销的逻辑）。让我们修改网站，使得点击登出链接时，所有指向 My Account 页面的打开窗口都跳转到网站首页。

在 app.js 中修改 `$(document).ready` 函数，如下所示：

```
$(document).ready(function() {
  $.getJSON("/events.json", renderEvents);

  if ("serviceWorker" in navigator) {
    $("#logout-button").click(function(event) {
      if (navigator.serviceWorker.controller) {
        event.preventDefault();
        navigator.serviceWorker.controller.postMessage(
          {action: "logout"}
        );
      }
    });
  }
});
```

代码检查了 service worker 的支持情况，如果可用，则给登出链接添加一个点击事件监听器。该事件监听器首先检查 service worker 是否在控制页面，如果是，就会阻止链接的默认行为，然后发送消息给 service worker，而不是跳转页面。

这是渐进增强的一个很好的示例。在一开始，登出链接类似于其他简单的 HTML（`<a href="/">Logout</a>`），并且功能齐全。然后，我们对它进行了增强，让它在支持 service worker 的浏览器下，支持多个窗口同时登出，而且不会破坏旧浏览器中的默认行为。

接下来，我们在 service worker 中添加监听这个消息的代码。

在 serviceworker.js 的结尾处，添加下列代码：

```
self.addEventListener("message", function(event) {
  var data = event.data;
  if (data.action === "logout") {
    self.clients.matchAll().then(function(clients) {
      clients.forEach(function(client) {
        if (client.url.includes("/my-account")) {
          client.postMessage(
            {action: "navigate", url: "/"}
          );
        }
      });
    });
  }
});
```

这段代码监听了 message 事件，从事件对象（event.data）中获取了消息数据，并决定如何操作。如果消息数据包含了名为 logout 的操作，监听器就会获取当前打开的所有 WindowClient，逐个遍历，并检查窗口的 URL 是否包含 /my-account。如果包含，就向这个窗口发送一条消息，其中包含了要采取的操作 navigate 以及操作的 URL /。



这个消息对象结构包含了要采取的操作以及额外的参数。这个结构完全是任意的，选择它是因为它适合用于这种情况。此处 navigate 对于浏览器来说并没有特定的含义，它只是我选择的一个字符串，描述了我希望应用所采取的操作。

接下来，我们要修改页面，监听 service worker 发送的消息。

在 app.js 中，修改 $(document).ready 函数，如下所示：

```
$(document).ready(function() {
  $.getJSON("/events.json", renderEvents);

  if ("serviceWorker" in navigator) {
    navigator.serviceWorker.addEventListener("message", function (event) {
      var data = event.data;
      if (data.action === "navigate") {
        window.location.href = data.url;
      }
    });

    $("#logout-button").click(function(event) {
      if (navigator.serviceWorker.controller) {
        event.preventDefault();
        navigator.serviceWorker.controller.postMessage(
          {action: "logout"}
        );
      }
    });
  }
});
```

这段代码新添加了一个事件监听器，监听了 service worker 的 message 事件。当这个事件监听器触发时，会获取消息内容，并检查消息中是否包含了值为 navigate 的 action 属性。如果包含，就将当前页面跳转到消息中指定的 URL。

就是这样！我们刚才渐进增强了一个简单的 HTML 链接，使得它不仅能跳转当前窗口，还能操作所有符合条件的其他窗口（即显示 My Account 页面的窗口）。

实现这一点的逻辑很简单。

如果 service worker 正在控制页面，则重写登出链接的默认操作，改为发送消息给 service worker，告诉它进行一个 logout 操作。与此同时，service worker 要监听这些消息，并在检测到这些消息时，向包含 /my-account URL 的所有受控制窗口发送消息，告诉它们采取 navigate 操作。页面也要监听这些消息，并且在检测到消息时，每一个窗口都要跳转到消息中包含的 URL。



请注意，在示例代码中，在添加事件监听器之前，我们没有检查 service worker 是否在控制页面。虽然只有当前由 service worker 控制的页面才能发送消息到 service worker，但是任何页面都可以为传入消息的事件添加事件监听器。

在 8.2 节中可以看到 service worker 向未受控制页面发送消息的例子。

8.6　从sync事件向页面传递消息

让我们把注意力转向本章开头提出的挑战。

在第 7 章中我们了解到，将事件从页面转移到 service worker 中，可使应用更具弹性和可靠。但是这暴露出一个新的困难。如果页面将发送消息、帖子点赞或者发起新预订这样的事件委托给了 sync 事件，我们如何在事件完成后更新 DOM 呢？既然我们知道了如何在 service worker 和页面之间发送消息，就拥有了解决这个问题所需的所有工具。

在 7.4 节中，我们将发起新预订的逻辑从 My Account 页面移动到了 sync 事件中。不幸的是，在 sync 事件成功完成时，我们并不能将消息传回给窗口。虽然我们可能使应用更具弹性了，但实际上，我们在用户体验上后退了一步。虽然在 sync 事件之前的代码确实在发起预订时及时更新了 DOM，但是新的同步代码要等到下次页面从网络请求更新时，才会更新预订状态。

让我们修复这个问题。

在 serviceworker.js 中更新 `syncReservations()` 函数，如下所示：

```
var syncReservations = function() {
  return getReservations("idx_status", "Sending").then(function(reservations) {
    return Promise.all(
      reservations.map(function(reservation) {
        var reservationUrl = createReservationUrl(reservation);
        return fetch(reservationUrl).then(function(response) {
          return response.json();
        }).then(function(newReservation) {
          return updateInObjectStore(
            "reservations",
            newReservation.id,
            newReservation
          ).then(function() {
            postReservationDetails(newReservation);
          });
        });
      })
    );
  });
};
```

新的 `syncReservations()` 函数包含了一处改进；在调用 `updateInObjectStore()` 之后，它还调用了 `postReservationDetails()`，传入了从网络接收到的新预订详情。

接下来在 serviceworker.js 中添加 postReservationDetails() 函数，放置在 syncReservations() 上方：

```
var postReservationDetails = function(reservation) {
  self.clients.matchAll({ includeUncontrolled: true }).then(function(clients) {
    clients.forEach(function(client) {
      client.postMessage(
        {action: "update-reservation", reservation: reservation}
      );
    });
  });
};
```

postReservationDetails() 的代码获取了 service worker 作用域内所有的客户端，对其进行迭代，然后向它们逐一发送消息。消息中包含了新预订的详情，并将它让浏览器采取的操作命名为 update-reservation。

最后，回到 app.js 中，更新早前添加的 message 事件监听器，让其处理这一类消息：

```
navigator.serviceWorker.addEventListener("message", function (event) {
  var data = event.data;
  if (data.action === "navigate") {
    window.location.href = data.url;
  } else if (data.action === "update-reservation") {
    updateReservationDisplay(data.reservation);
  }
});
```

这段代码增添了另一个判断条件，寻找操作为 update-reservation 的消息。当检测到这类消息时，代码会调用 updateReservationDisplay() 函数，并传入消息中包含的新预订详情。updateReservationDisplay() 可以在 my-account.js 中找到，这个方法接收一个预订对象，并在 DOM 中更新该预订的详情。

在第 7 章中，我们将预订逻辑移动到了 service worker 中。现在，只需几条额外的命令，就可以将这些操作的结果传回页面，并更新显示。这个闭环就完成了。

8.7 小结

本章探索了如何使用 postMessage() 在 service worker 及其控制的窗口之间进行通信。我们能够用 sync 事件中更新的预订数据增强应用的 UI，并能在不同的窗口之间同步登录状态。

在第 11 章中，我们将结合本章所学的知识，进一步提升用户体验。例如，当应用已经缓存了离线使用所需的资源之后，我们可以通过从 service worker 的 install 事件向页面发送消息，告知用户。

不过，首先我们要探讨渐进式 Web 应用最令人激动的两项新特性。

第 9 章

可安装的Web应用：占领主屏先机

我们已经取得了很多成就，并学会了如何在 Web 上做许多以前无法想象的事情，但是到目前为止，我们仍然牢牢扎根在浏览器领域。本章，我们将最终超越浏览器，开辟一个新的领域，这个领域曾经是原生应用专属的。

我们将看到如何在用户主屏上占领先机，并构建出可以安装在用户设备上的 Web 应用。当用户访问这些 Web 应用时，浏览器会自动提示用户将它们安装到设备的主屏幕上。这些 Web 应用可以在全屏模式下启动（没有任何浏览器本身的界面），使得它们看起来和原生应用无异，并且可以锁定在某个屏幕方向上（即横屏或者竖屏模式），等等（见图 9-1）。

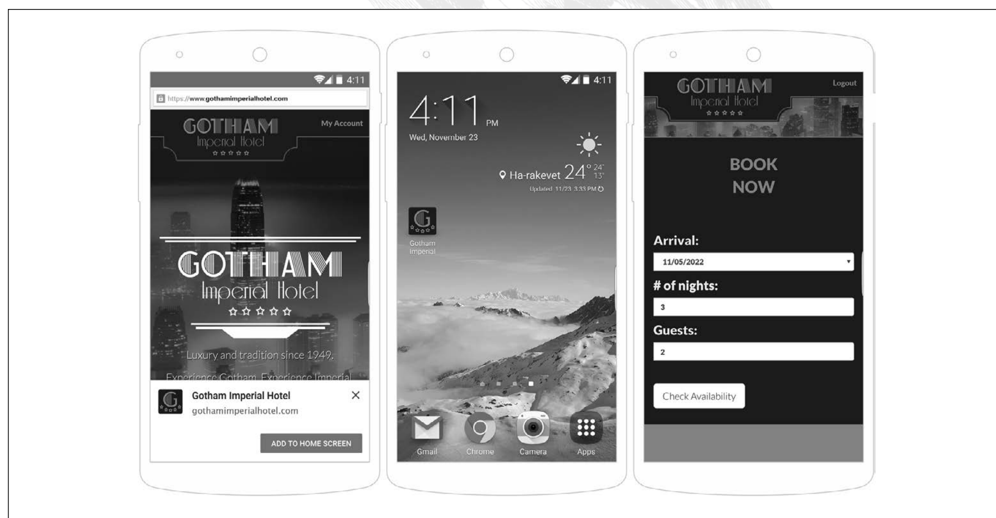


图 9-1：Chrome 的 Web 应用安装过程

9.1 可安装的Web应用

本章开头承诺的功能看起来很神奇，但是其实现却异常简单。实际上，只需要三个步骤：

(1) 注册 service worker；
(2) 创建 Web 应用清单文件；
(3) 在 Web 应用中，添加这个清单的链接。

鉴于我们已经在 Web 应用中注册了一个 service worker，我们已经完成了三分之一的工作。下面来完成剩下的两步。

首先，创建一份 Web 应用清单（web app manifest）。

这个清单是一个简单的 JSON 文件，描述了 Web 应用应该如何启动和表现及其外观。就是这么简单。

在开始前，要通过在命令行中运行以下命令，确保代码处于上一章结束时的状态：

```
git reset --hard
git checkout ch09-start
```

接下来，在哥谭帝国酒店项目的 `public` 目录中，创建一个名为 manifest.json 的文件，内容如下：

```
{
  "short_name": "Gotham Imperial",
  "name": "Gotham Imperial Hotel",
  "description": "Book your next stay, manage reservations, and explore Gotham",
  "start_url": "/my-account?utm_source=pwa",
  "scope": "/",
  "display": "fullscreen",
  "icons": [
    {
      "src": "/img/app-icon-192.png",
      "type": "image/png",
      "sizes": "192x192"
    },
    {
      "src": "/img/app-icon-512.png",
      "type": "image/png",
      "sizes": "512x512"
    }
  ],
  "theme_color": "#242424",
  "background_color": "#242424"
}
```

虽然 manifest.json 中的内容不言自明，但我们将在 9.3 节中深入研究清单文件的细节。

接下来，在 index.html 和 my-account.html 的头部添加下列 HTML 标签，让浏览器知道这个网站有可用的清单文件。

```
<link rel="manifest" href="/manifest.json">
```

就是这样！

现在一切都取决于浏览器了。

9.2　浏览器如何决定何时显示应用安装横条

当浏览器确定一个网站可以安装，并且用户可能对于该网站有足够的兴趣，并希望在主屏上放置快捷方式时，就会触发一个 Web 应用安装横条（如图 9-2 所示）。[1]

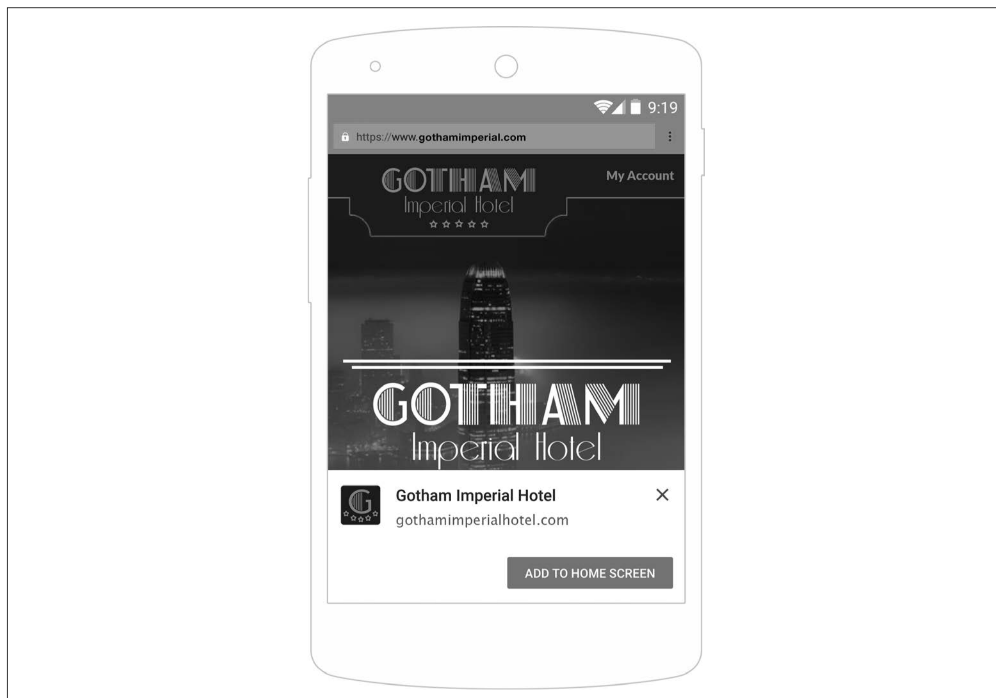


图 9-2：Chrome 中的 Web 应用安装横条

浏览器只有在它认为应用满足特定的最低标准，能够提供类似于原生应用的体验，值得放置在用户主屏时，才会在网站上显示 Web 应用安装横条。

在编写本书时，这些标准如下：

(1) 网站提供 HTTPS 服务；
(2) 网站注册了 service worker；
(3) 网站拥有一份 Web 应用清单，其中至少包含了四个必填字段（详见 9.3 节）。

此外，浏览器只有在认为用户可能足够在乎这个 Web 应用，希望在主屏上放置一个永久快捷方式时，才会显示 Web 应用安装横条。至于如何确定这一点，不同浏览器以及不同浏览器版本之间都会有所不同。举个例子，最初启用这个功能时，当用户在两周之内有两天访问了应用时，Opera 和 Chrome 就会显示安装横条。后来，这些启发式方法已经被修改了，以增加安装横条的显示频率，并且目前仍然在不断被各浏览器厂商调整，以微调用户的体验。

注 1：本书中提到的安装横条指的是不同种类的安装提示，其中包括 Chrome 和 Opera 中的 Web 应用安装横条、三星浏览器中的徽章等。

总而言之，如果满足以下条件，浏览器就会显示安装横条：

```
if (
  Web应用提供HTTPS &&
  Web应用注册了service worker &&
  应用拥有有效的清单文件，其中包含了所有必需的属性&&
  清单文件在用户访问的页面中有链接&&
  浏览器认为用户对于这款应用有持久的兴趣&&
  这款应用的安装横条在过去没有被显示和拒绝过
) then {
  显示Web应用安装横条
}
```

9.3 剖析Web应用清单

在继续之前，我们先探索 Web 应用清单的格式。

任何有效的 JSON 文件都可以成为清单文件，但是要触发 Web 应用安装横条，清单文件就必须至少包含以下属性。

`name`与/或`short_name`

清单文件必须包含 `name` 或者 `short_name` 属性，或者（最好）两者都包含。

`name` 是应用的全名。当空间足够长时，就会使用这个字段作为显示名称，例如显示在应用安装横条以及应用的启动屏幕上。

如果应用名称特别长，那么在没有足够空间显示全名的情况下，`short_name` 可以作为短名的备选方案。短名会用在应用图标的旁边、任务管理器中，以及任何不适合显示全名的地方。请确保 `short_name` 不超过 15 个字符，这样它才不会在主屏上被截断。

让我们来看一个例子。如果你的应用全名相对较短，可以自由选择提供 `short_name` 和 `name` 中的一个，而忽略另一个参数。如果应用名称比较长（例如 Gotham Imperial Hotel），那么就要同时提供全名和备选的短名（例如 Gotham Imperial），才可以确保设备不会自动截取应用名称（显示“Gotham Imperial”比起显示“Gotham Imperial Hot...”好多了）。

`start_url`

当用户点击图标时，打开的 URL。可以是根域名，也可以是内部页面。

对于哥谭帝国酒店，当用户从主屏启动应用时，我们使用了 My Account 页作为第一个显示的页面，而不是首页。我们还在查询字符串中添加了一个 `utm_source=pwa` 标签，以便分析软件用它来跟踪从主屏启动的访客。如果你要在你的应用中做同样的事情，请确保 service worker 知道如何匹配查询字符串中带有或不带 `utm_source` 的情况（5.7 节中的代码可以正确匹配这些请求，因为它使用了 `pathname` 参数，没有包括查询字符串的匹配）。

`icon`

包含了一个或多个对象的数组，每个对象描述了一个 Web 应用可以使用的图标。其中每个对象都会包含以下属性：`src`（图标的绝对路径或者相对路径）、`type`（文件类型）

和 sizes（图片的像素尺寸）。要触发 Web 应用安装横条，清单中至少要包含一个图标，尺寸至少是 144 像素 ×144 像素。

由于每个设备都会根据设备分辨率，从这个数组中选择最佳的图标尺寸，因此建议至少包含 192×192 的图标和 512×512 的图标，以覆盖大多数的设备和用途。

display

控制应用启动时的显示模式（见图 9-3）。

可能的值如下。

- browser——在浏览器中打开应用。
- standalone——打开应用时不显示浏览器栏（不显示浏览器界面，例如地址栏）。
- fullscreen——打开应用时不显示浏览器栏和设备栏（例如在安卓设备上，这意味着同时隐藏浏览器界面和屏幕顶部的状态栏）。

  使用桌面应用的说法，可以把 standalone 和 fullscreen 分别视为最大化或者全屏模式的应用，而 browser 的行为就和在浏览器中点击任何链接是一致的。

  要显示 Web 应用安装横条，display 属性必须设置为 fullscreen 或者 standalone。



图 9-3：从左到右：browser 模式、standalone 模式和 fullscreen 模式

除了上面描述的属性最小集外，Web 应用清单还支持下列属性。

description

应用的描述。

orientation

允许你强制指定某个屏幕方向。如果你的应用布局在竖屏或者横屏时表现更加，那么这会非常有用。例如，许多游戏会强制采用横屏模式，而重文本类应用通常更倾向于竖屏

模式（见图 9-4）。

以下是 `orientation` 最常见的可能取值：

- `landscape`
- `portrait`
- `auto`

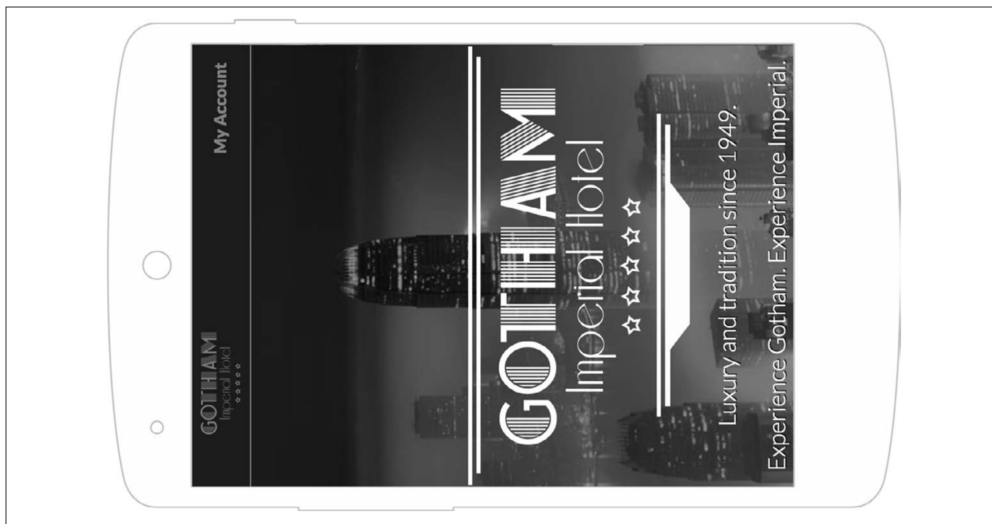


图 9-4：被锁定为竖屏的 Web 应用

`theme_color`

主题颜色可以让浏览器和设备调整 UI 以匹配你的网站（见图 9-5）。这个颜色的选择会影响浏览器地址栏颜色、任务切换器中的应用颜色，甚至是设备状态栏的颜色。

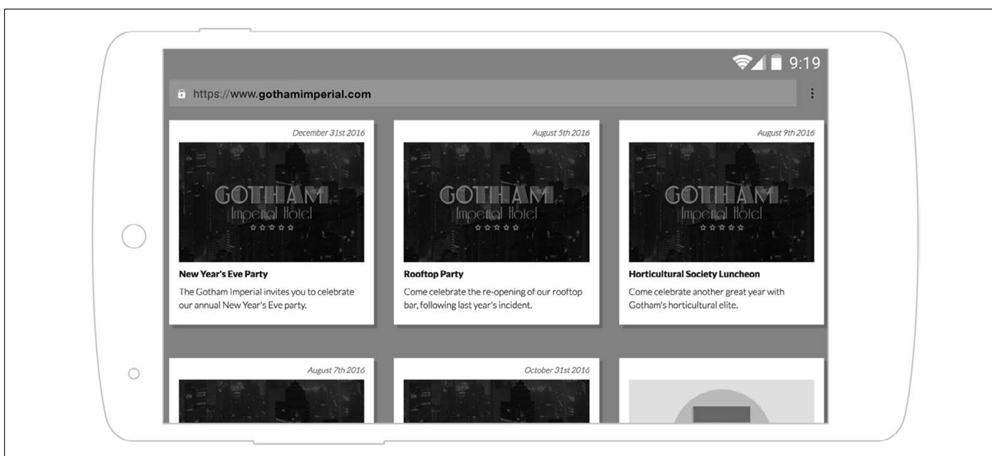


图 9-5：带有主题颜色的网站和手机界面完美融合

主题颜色也可以通过页面的 meta 标签进行设置（例如：`<meta name="theme-color" content="#2196F3">`）。如果页面带有 `theme-color` 的 meta 标签，则该设置会覆盖清单中的 `theme_color` 设置。请注意，虽然 meta 标签可以让你设置或者覆盖单个页面的主题颜色，但是清单文件中的 `theme_color` 设置是会影响整个应用的。

`background_color`

设置应用启动画面的颜色以及应用加载时的背景色。一旦加载后，页面中定义的任何背景色（通过样式表或者内联 HTML 标签设置）都会覆盖这一设置；但是，通过将其设置为与页面背景色相同的颜色，就可以实现从页面启动的瞬间到完全渲染之间的平滑过渡。如果不设置这一颜色，页面就会从白色背景启动，随后被页面的背景色替换。

`scope`

定义了应用的作用域。当用户处于一个 `full-screen/standalone` 模式的应用中，并跳转到这个作用域下的另一个 URL 时，这个 URL 也会在 `full-screen/standalone` 模式下打开。然而，如果用户点击的链接将他带出了作用域范围外，链接就会在常规浏览器窗口中打开。

例如，如果我们设置了 `"scope": "/my-account/"`，当用户在这个作用域内跳转时（例如 `/my-acount/talater` 或者 `/my-account?sort=date`），会留在应用中。但是一旦用户点击了这个范围外的链接（例如 `/index.html` 或者 `https://pwabook.com`），页面就会在浏览器中打开。

在某些浏览器中，`scope` 还会用来设置安卓系统的 Intent Filter。当一个 Web 应用被安装并且设置了 `scope` 时，任何指向应用作用域内页面的链接都会启动这款应用，而不是直接在浏览器中打开。例如，如果用户之前已经安装了我们的渐进式 Web 应用，然后从一个旅游评论网站点击了链接 https://www.GothamImperial.com/my-account，那么该链接就会启动我们的应用，而不是在浏览器中展示页面。

`dir`

显示 `name`、`short_name` 和 `description` 参数文本的方向。默认情况下适配浏览器的语言设置，但是也可以设置为以下值之一。

- `ltr`——从左到右的语言，例如英语和葡萄牙语
- `rtl`——从右到左的语言，例如希伯来语和阿拉伯语
- `auto`——使用浏览器的语言设置

`lang`

指定 `name`、`short_name` 和 `description` 参数文本的主要语言。

可以和 `dir` 参数一起用来保证任何语言的文本正确显示，包括从右到左的语言。

`prefer_related_applications`

如果你还有一款原生应用，并且你更喜欢浏览器提供原生应用，而不是你闪亮的新渐进式 Web 应用，那么可以把 `prefer_related_applications` 设置为 `true`。

当设置为 `true`，并且把当前平台下的原生应用列举在 `related_applications` 中时，网

页会显示原生应用的安装横条而不是 Web 应用的安装横条。除了不依赖于 service worker 之外，显示原生应用安装横条的要求和显示 Web 应用安装横条的要求是一致的。

`related_applications`

这个参数接收一个“应用对象”的数组。每个对象中包含了一个 `platform` 平台参数（例如 `play`、`itunes`）、一个 `url` 参数（表明应用可以在哪里获取），还有 `id` 参数（用来表示特定平台中的标识）。

下面的示例定义了关联的 Android 和 iPhone 应用，并告诉浏览器优先显示原生应用安装横条，而不是 Web 应用安装横条：

```
"related_applications": [
  {
    "platform": "play",
    "url": "https://play.google.com/store/apps/details?id=com.goth.app",
    "id": "com.goth.app"
  }, {
    "platform": "itunes",
    "url": "https://itunes.apple.com/app/gotham-imperial/id1234"
  }],
"prefer_related_applications": true
```

9.4 各端兼容性

你安装的应用图标，在 Android 中的显示方式与 Windows 8、Windows 10 中的显示方式大不相同，和较新的带 Touch Bar 的 MacBook Pro 相比也大不相同。即使是在单个平台下，图标也可能根据屏幕分辨率的不同而变化很大。

每个平台、浏览器、操作系统和设备显示应用和图标的方式都不一样。

坦率地说，这是个不断变化的雷区。

试图跟上变化，并在本书中涵盖每个平台的要求是不现实的。幸运的是，有很多很棒的在线工具可以帮助你优雅地处理这些复杂性；你可以在 https://pwabook.com/appicons 中找到这份列表。

除了本章早些时候添加的清单文件外，哥谭帝国酒店已经配置了一些在不同平台上显示图标所需的更重要的设置。你可以在 index.html 的 `<HEAD>` 标签中看到这些代码：

```
<link rel="apple-touch-icon" sizes="180x180" href="/img/apple-touch-icon.png">
<link rel="icon" type="image/png" href="/img/favicon-32x32.png" sizes="32x32">
<link rel="icon" type="image/png" href="/img/favicon-16x16.png" sizes="16x16">
<link rel="shortcut icon" href="/favicon.ico">
<link rel="mask-icon" href="/img/safari-pinned-tab.svg" color="#a3915e">
<meta name="msapplication-config" content="/browserconfig.xml">
<meta name="theme-color" content="#242424">
```

这些设置中包含了添加到 iPhone 的主屏快捷方式图标（`apple-touch-icon`）、可信的 `favicon`（在浏览器标签和书签中显示）、Safari 固定标签图标（`mask-icon`），以及微软的应用配置文件（`msapplication-config`）链接，这个配置文件可以决定 Windows 设备上的应用外观。此外，我们还为较旧的浏览器定义了 `theme-color`，而不是从清单文件中读取。

9.5 小结

几年前，浏览器菜单中隐藏很深的“添加到主屏快捷方式”选项，现在已经演变成了可安装的 Web 应用。

这些应用结合了原生应用的所有好处，同时又避免了许多缺点，其中包括野蛮的安装模型，将其替换成逐步在用户设备上立足的方式。

但是权力越大责任越大。如果想让应用可以和原生应用比肩，就需要考虑给予用户的体验。我们将在第 11 章深入探讨这一点。

但首先，我们会在第 10 章中探索推送消息，进一步脱离浏览器。

第 10 章
推送通知

很少有（如果有的话）功能像推送通知那样，成为原生应用和 Web 应用之间的主要鸿沟。

推送通知让用户能够选择获得他们所关注的应用的更新，并及时更新他们所需的内容和数据。你能想象到使用一款不提供通知的即时消息应用的场景吗？

作为开发者，推送通知可以让我们改善应用的用户体验，并因此提高使用率。对于应用的采用和成功来说，它们可能比其他任何因素都更重要。

对于企业而言，能够重新获得用户，并将他们一次又一次地带回应用中，是提高每次应用安装的价值的关键。这使得企业可以投入更多的资金来获取用户，同时仍保持投资的正回报。

毫不夸张地说，推送通知一直是原生应用成功的最大驱动因素之一。

但是，既然 Web 已经能够充分获得推送通知的能力，我们终于可以推翻本章开头的陈述了：很少有（如果有的话）功能能像推送通知那样，给 Web 应用带来如此巨大的影响。

10.1 推送通知的生命周期

我们从第 1 章开始就一直在讨论推送通知，现在终于是时候说这句话了：

> 事实上，推送通知包含了两个概念。

推送通知实际上包括了两件独立的事情：使用 Push API 发送消息，使用 Notification API 显示通知。

10.1.1 Notification API

Notification API 可以让网页或 service worker 创建并控制系统通知的显示。

通知会显示在浏览器的外部（在设备的 UI 上），因此通知是存在于任何单个浏览器窗口或者标签页的上下文之外的。由于通知不依赖于任何浏览器窗口或者标签页，甚至可以在用户离开网站之后再创建它们。

在你向用户显示通知之前，需要首先请求用户的许可。

整个过程很简单，就如下面这个功能完备的代码示例所展示的那样：

```
Notification.requestPermission().then(function(permission){
  if (permission === "granted") {
    new Notification("Shiny");
  }
});
```

只需用这段示例代码就可以请求显示通知的权限。然后，如果权限被授予（granted），就创建一个标题为 Shiny 的通知。就这么简单。

在本章后面，我们将会添加按钮、图标，甚至让通知使用《星球大战》主题振动用户的手机。

10.1.2 Push API

Push API 允许用户同意应用推送消息，让服务器可以随时推送消息到浏览器。这些消息会由 service worker 监听并处理，甚至在用户离开应用后也可以进行操作。最常见的操作方式就是向用户显示通知。

这给应用带来了巨大的能量。一旦你可以在任何时候向用户的设备发送消息，你就有可能用无尽的消息来骚扰他。你甚至可以通过每隔几秒向 service worker 发送消息，然后将一些影响数据响应发回服务器，来静默地跟踪用户的行为。

为了确保 Push API 不会被这样滥用，所有的推送消息都要通过中心消息服务器。中心服务器由浏览器供应商维护，它会为你跟踪所有用户的订阅。它确保推送消息不会被滥用，且用户不会被骚扰。即使用户在你发送消息时无法触达，它也能确保消息被到达。

你和用户之间的中间人，以及确保只有你的服务器可以给用户发送消息所需的全部加密过程，让学习曲线变得有些陡峭。我们将把这个过程分成四个步骤，逐一讲解。

前两个步骤是用户订阅推送消息，以及将推送的详情保存到你的服务器上。这两个步骤每个用户只需执行一次。

后两个步骤——从服务器发送消息以及在浏览器中进行操作——会在每次你想要发送消息给用户时发生。这可以在创建订阅后立即进行，也可以在一周后才进行。

首先看看前两个步骤（见图 10-1）。

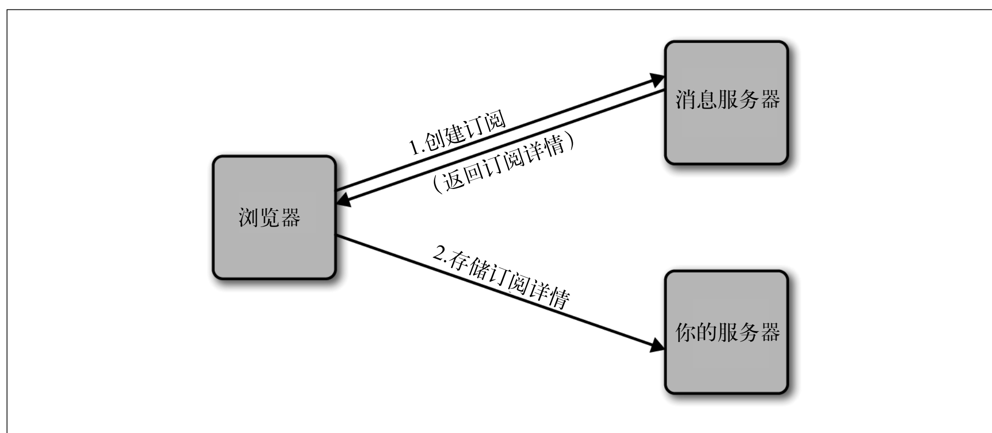


图 10-1：创建并存储推送订阅

首先，Web 页面使用了 Push API 来调用 `subscribe()`。这将会调用中心消息服务器，该服务器会存储新订阅的详情，并将详情返回给页面。接下来，页面可以将订阅详情发送给你的服务器，服务器可以将其存储下来，以供将来使用。你将经常需要把这些订阅细节保存在数据库中，也许是保存在你用来保存其他用户详情的同一个表或者对象存储中。

接下来是每次你要发送消息时需采取的后两个步骤（见图 10-2）。

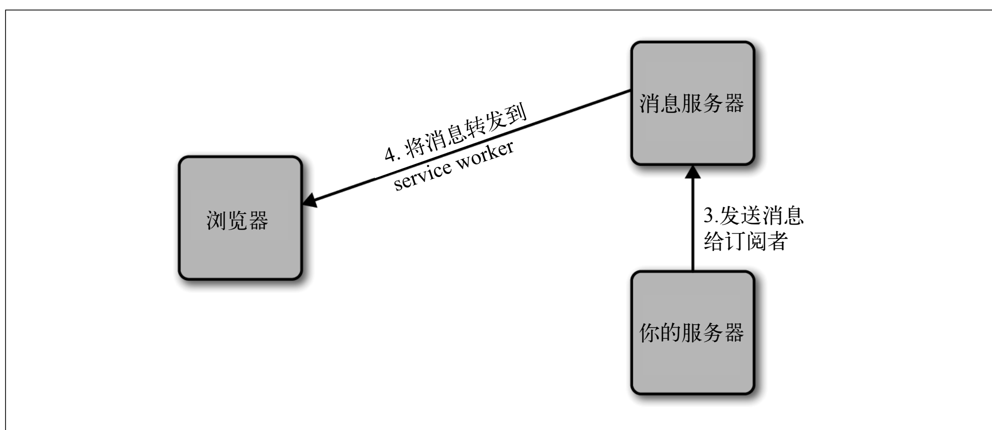


图 10-2：从服务器发送推送消息

当你决定要发送消息时，服务器要先获取它之前存储的订阅详情（步骤 2），然后使用这些数据把消息发送到消息服务器。然后，消息服务器将消息转发到用户的浏览器。最后，用户浏览器中注册的 service worker 接收到消息，阅读其内容，并决定如何处理。

最后一个注意事项是：创建新的推送订阅（步骤 1）需要用户的许可。幸运的是，它使用的权限和显示通知所需的权限相同，所以你只需要请求一次权限，就能显示通知和发送推送消息。

10.1.3 Push+Notification

让我们综合以上内容，看看向用户发送推送通知的整个过程：

(1) 页面向用户请求显示通知的权限，用户授权；
(2) 页面和中央消息服务器通信，要求服务器为这个用户创建一个新的订阅；
(3) 消息服务器返回新的订阅详情对象作为响应；
(4) 页面将订阅详情发送给服务器；
(5) 服务器将订阅详情储存起来，以供将来使用；
(6) 时间流逝，季节变化，需要发送新的通知；
(7) 服务器使用订阅详情，通过消息服务器将消息发送给用户；
(8) 消息服务器将消息转发给用户的浏览器；
(9) service worker 的 push 事件监听器收到消息；
(10) service worker 显示通知，其中包含了消息内容。



推送通知的浏览器支持度

在大部分现代桌面浏览器中，可以从**活动**的窗口创建简单的通知，如本章前面所示。

接收推送消息并显示通知，需要 service worker、Notification API 和 Push API 的支持。

在本书编写时，Firefox、Chrome、Chrome for Android、Samsung Internet 和 Opera 已经支持，Edge 浏览器正在开发支持中。

在本章介绍的 API 完成之前，苹果公司就已经为 Safari 用户创建了发送通知的专用 API。你可以在 Apple 开发者网站上阅读更多相关信息。

10.2 创建通知

现在我们对于推送通知有了理论上的理解，让我们开始编写第一个通知。

和往常一样，首先在命令行中运行以下命令，以确保代码处于上一章结束时的状态：

```
git reset --hard
git checkout ch10-start
```

10.2.1 请求通知权限

正如我们在 10.1 节中看到的那样，在向用户显示通知之前，首先要获得用户的许可。

你可以通过检查 `Notification.permission` 的值，判断当前网站是否具有创建通知的权限。如果当前页面拥有显示通知的权限，`permission` 的值会等于 `granted`；如果用户尚未决定，值会是 `default`；如果用户拒绝了权限请求，那么值会等于 `denied`：

```
if (Notification.permission === "granted") {
  console.log("Notification permission was granted");
}
```

如果你还没有获得权限，可以通过调用 Notification API 的 `requestPermission()` 方法，向用户请求权限：

```
Notification.requestPermission();
```

这样做会在浏览器界面显示请求权限（见图 10-3）。

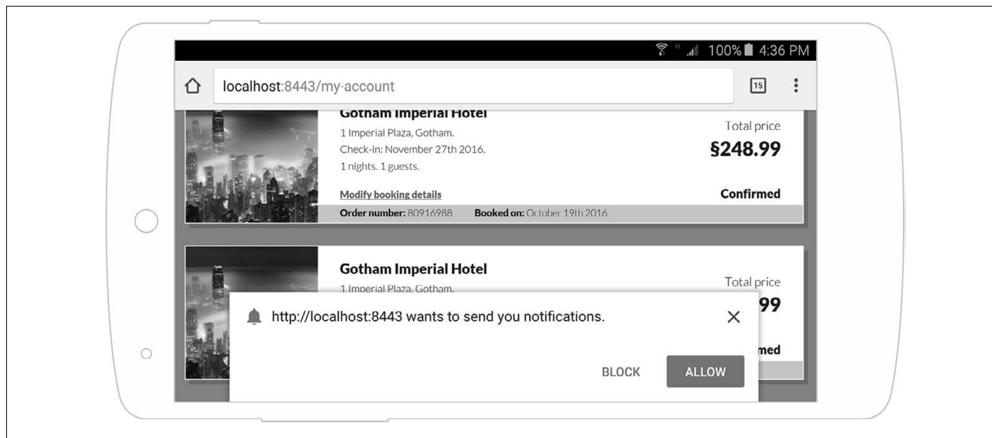


图 10-3：通知权限的对话框

`requestPermission` 方法会返回一个 promise，当用户（或者浏览器）做出许可决定后，promise 会完成。需要重点记住的是，这个 promise 即使在用户拒绝许可，或者浏览器自动阻止了权限请求许可时，也会完成。所以，在请求权限后、尝试创建通知之前，总是要检查许可的当前状态，这一点非常重要。

```
Notification.requestPermission().then(function(permission) {
  if (permission === "granted") {
    console.log("Notification permission granted");
  }
});
```

在 `requestPermission()` 返回的 promise 中，permission 参数可以是下列值中的任意一个。

`granted`

当前页面具有显示通知的权限。这意味着两种可能：

(1) `requestPermission()` 被调用后，显示权限许可对话框，用户选择同意；
(2) `requestPermission()` 被调用，但由于用户之前已经授权过，所以不需再显示权限许可对话框。

`denied`

当前页面不具有显示通知的权限。这意味着两种可能：

(1) `requestPermission()` 被调用后，显示权限许可对话框，但用户选择拒绝；
(2) `requestPermission()` 被调用，但由于用户之前已经拒绝过，所以不需再显示权限许可对话框。

`default`

当前页面不具有显示通知的权限。这种情况只有一种可能：

- `requestPermission()` 被调用后，显示权限许可对话框，但用户没有做出选择，直接关闭了对话框。

将所有内容放在一起，我们就得到了下面的代码：

```
if (Notification.permission === "granted") {
  showNotification();
} else if (Notification.permission === "denied") {
  console.log("Can't show notification");
} else if (Notification.permission === "default") {
  Notification.requestPermission().then(function(permission) {
    if (permission === "granted") {
      showNotification();
    } else if (Notification.permission === "denied") {
      console.log("Can't show notification");
    } else if (Notification.permission === "default") {
      console.log("Can't show notification, but can ask for permission again.");
    }
  });
}
```

虽然在许多情况下，在决定如何处理之前，你需要先使用 `Notification.permission` 检查当前的权限状态（如上述代码），但在其他情况下，你只需要调用 `requestPermission()`，并相信浏览器只会在必要时显示权限许可对话框。这样我们可以将上述的代码示例简化如下：

```
Notification.requestPermission().then(function(permission) {
  if (permission === "granted") {
    showNotification();
  } else if (Notification.permission === "denied") {
    console.log("Can't show notification");
  } else if (Notification.permission === "default") {
    console.log("Can't show notification, but can ask for permission again.");
  }
});
```

同源策略

用户的选择是根据同源策略保存的。换句话说，一旦用户授予权限，你就可以在应用中同源的任何页面下创建新的通知。

如果两个 Web 页面的 URI 协议（例如 HTTPS、HTTP）、主机名（例如 www.talater.com）和端口号都一致，那么这两个页面就是同源的。例如，在 https://www.talater.com/annyang 授予的权限，可以在 https://www.talater.com/upup 上使用，但是不能在 http://www.talater.com/annyang（使用 HTTP，而权限授予在 HTTPS 协议上）或者 https://www.talater.com:8443/（端口号不一致）上使用。

10.2.2 显示通知

一旦获得了用户的许可，创建通知就是创建一个新的 `Notification` 对象。让我们来试一试：

```
Notification.requestPermission().then(function(permission) {
  if (permission === "granted") {
    new Notification("Shiny");
  }
});
```

如果你在浏览器控制台中运行这段代码，应该会向你请求显示通知的权限，紧接着显示一条标题名为 `Shiny` 的简单通知（见图 10-4）。

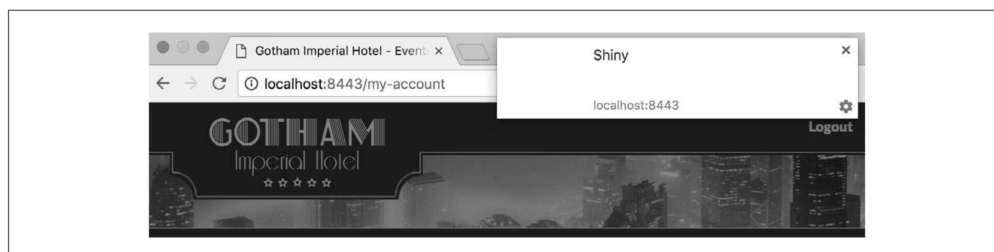


图 10-4：可能是最简单的桌面通知



修改通知设置权限

如果你没有看到权限许可对话框，可能是因为之前你拒绝或者授权过这个站点的通知。

一旦你在权限许可对话框中做出了选择，浏览器就会记住选择，并且不会在这个源中再次显示通知权限许可对话框。在开发过程中，可能你需要不时地重设这个设置。

在桌面版 Chrome 中，可以通过点击地址栏网站 URL 左侧的图标，并更改通知的设置来完成。在安卓版 Chrome 中，可以打开浏览器菜单，选择设置，点击网站设置，然后找到同样的设置项。

不幸的是，虽然上述代码在桌面端可以正常运行，但是在移动设备上是不能正常工作的。要理解原因，先要考虑移动端的通知是如何表现的。当页面创建通知时，它会在浏览器外部渲染，也就是在操作系统层面上渲染。通知可能会维持可见，用户可能会在离开网站很久之后才与它进行交互。为了确保我们可以捕捉到用户与通知的交互，通知需要“逗留”在更高的级别——service worker 中。

要创建可以同时在桌面端和移动端工作的通知，就需要通过 service worker 来创建。幸运的是，使用 service worker 的 `registration` 对象，你甚至不需要修改 service worker 的代码，就能轻松地在页面中完成这一操作。

只需要对代码稍作修改，我们就可以在 service worker 的 `registration` 对象上调用 service worker。它接收的参数和 `Notification` 对象方法完全相同。

```
Notification.requestPermission().then(function(permission) {
  if (permission === "granted") {
    navigator.serviceWorker.ready.then(function(registration) {
      registration.showNotification("Shiny");
    });
  }
});
```

这种移动端友好的语法在移动设备和桌面都可以良好地运行（见图 10-5）。从这一点上讲，我们的代码只需要使用这种语法即可。在你自己的应用中，可能你会希望同时使用两种语法，以同时支持现代浏览器和不支持 service worker 的浏览器。

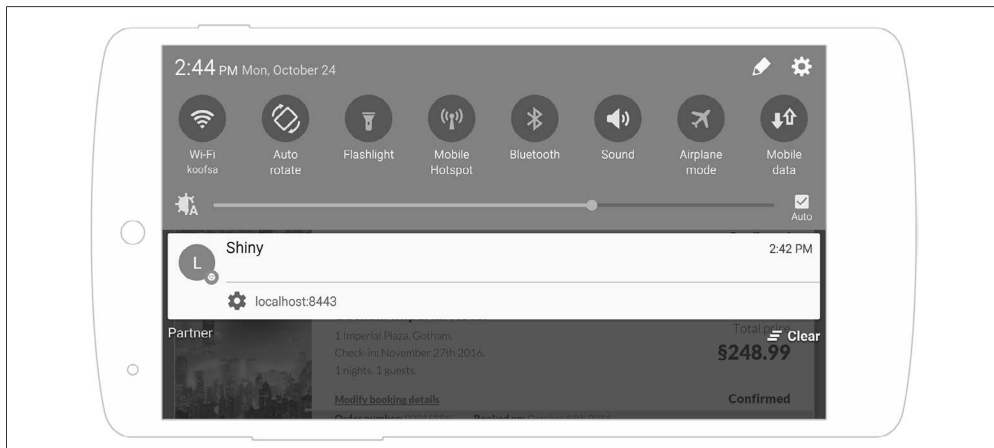


图 10-5：可能是最简单的移动端通知

现在我们已经知道如何创建一个简单的通知，下面来看看一些额外的选项，它们可以改进我们的通知。

```
navigator.serviceWorker.ready.then(function(registration) {
  registration.showNotification("Quick Poll", {
    body: "Are progressive web apps awesome?",
    icon: "/img/reservation-gih.jpg",
    badge: "/img/icon-hotel.png",
    tag: "awesome-notification",
    actions: [
      {action: "confirm1", title: "Yes", icon: "/img/icon-confirm.png"},
      {action: "confirm2", title: "Hell Yes", icon: "/img/icon-cal.png"}
    ],
    vibrate:[500,110,500,110,450,110,200,110,170,40,450,110,200,110,170,40,500]
  });
});
```

更新后的代码演示了 `showNotification()` 如何接收一个可选的第二参数，其中包含了一个选项对象。这些选项可以用来进一步定制和修改通知的行为。

以下是在创建通知时，你可以使用的所有选项。`registration.showNotification()` 和 `new Notification()` 这两种用法都支持这些选项。

`body`

通知正文中的文本内容。

`icon`

将在通知中显示的图片 URL 地址（在图 10-6 中对应城市的图片）。

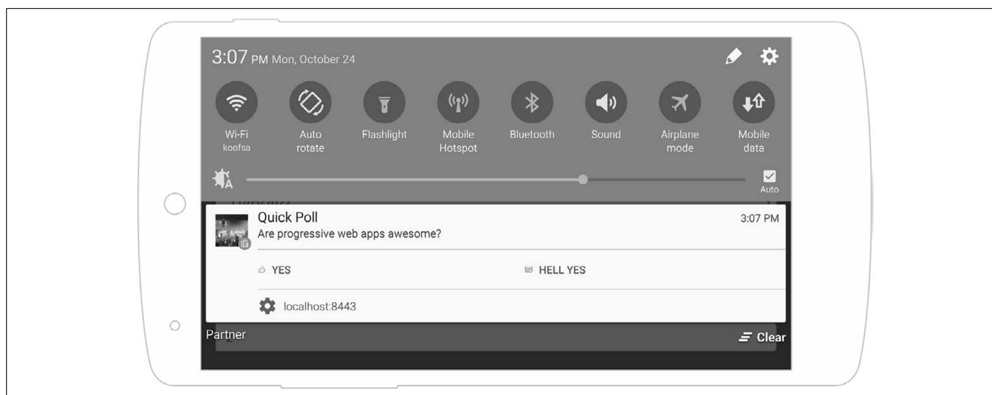


图 10-6：丰富移动端推送

`badge`

用来代表发送通知的应用的图片 URL，或者是代表应用发送的通知类别。例如，消息应用可能总是使用它的徽标作为所有通知的标记，也可能会选用不同的图标来代表不同的通知，例如：新消息通知使用一种图标，而用户名被提及时使用另一种图标。当没有显示整个通知的空间时，或者在通知内部（如图 10-6 中图标的右下角所示），都有可能会显示这个标记。

`actions`

通过传入一个操作对象数组，你可以给通知添加两个按钮，让用户可以直接在通知上执行操作。这可以让用户快速启动你的 Web 应用，甚至可以在不打开应用的情况下，直接从通知中快速进行操作。例如，消息应用中的新消息通知可以包含一个点赞按钮和一个回复按钮。点赞按钮可以在不打开应用的情况下工作，而回复按钮需要打开消息应用并显示合适的页面。我们会在 10.5 节中进一步了解这些操作。

`vibrate`

对于支持振动的设备，你可以自定义振动模式，使用这个模式提醒用户收到了这个新通知。`vibrate` 接收一个整型数组，每个数字以毫秒数为单位，描述了振动和暂停的时间。例如，`[200,100,300]` 表示振动 200 毫秒，暂停 100 毫秒，然后再振动 300 毫秒。在上述代码示例中，振动设置会播放《帝国进行曲》的节奏。

`tag`

表示这个通知的唯一标识符。如果这个标签等于当前正在显示的通知的标签，那么新通知会静默替代旧通知。这样做通常比创建多个通知来打扰用户更加可取。例如，如果用

户在我们的消息应用中有一条未读消息，我们可能想要在通知中包含消息文本。如果在新通知发送之前已经有五条消息到达了，那么将通知内容改为“你有 6 条新消息”，会比起展示 6 条通知更加合理。

下面的代码展示了如何创建一条通知，并且每隔一秒静默更新一条新通知，其中包含了不同的文本。这样做可以有效地实现一个带有计时器的通知：

```
navigator.serviceWorker.ready.then(function(registration) {
  var count = 1;
  var createNotification = function() {
    registration.showNotification("Counter", {
      body: count,
      tag: "counter-notification"
    });
    count += 1;
  };
  setInterval(createNotification, 1000);
});
```

如果你将标签删除，或者在每次迭代时修改标签，那么浏览器就会创建多条通知。

`renotify`

正如我们刚才看到的，如果使用相同的标签来更新现有的通知，则新的通知会悄然取代旧的通知。通过设置 `renotify` 属性为 `true`，你就可以在更新通知时，迫使设备吸引用户的注意（在移动设备上，这是通过再次振动手机完成的）。

`data`

可以用来附加任何想要伴随通知发送的数据。在本章的后面，我们会看到如何对通知事件做出响应，并获取这些数据（参见 10.5 节）。

`dir`

定义了在通知中显示文本的方向。默认情况下，它采用的是浏览器语言设置，但是也可以强制设置为 `rtl`（用于从右到左的语言，例如阿拉伯语和希伯来语）或者是 `ltr`（用于从左到右的语言，例如英语和葡萄牙语）。

`lang`

通知文本的主要语言。例如，`en-US` 对应美式英语，而 `pt-BR` 对应巴西葡萄牙语。

`noscreen`

一个布尔值，用来指定设备的屏幕是否会被这个通知打开。如果设置为 `true`，那么屏幕就不会被打开。在本书编写时，还没有任何浏览器支持这一属性，使用默认值 `false`。

`silent`

一个布尔值，用来指定通知是否静默（即没有振动或者声音）。在本书编写时，还没有任何浏览器支持这一属性，使用默认值 `false`（非静默）。

`sound`

在创建通知时，用于播放的音频文件 URL。在本书编写时，还没有任何浏览器支持这一属性。



通知试验场

如果你想要尝试使用通知，可以用你喜欢的代码编辑器打开 /public/notifications.html，并在 `<script>` 标签中进行修改。接下来，在开发服务器运行的情况下（参见 2.4 节），在浏览器中打开 http://localhost:8443/notifications.html 即可。

10.2.3　为哥谭帝国酒店添加通知支持

让我们继续为哥谭帝国酒店 Web 应用添加通知。我们的目标是：当用户在哥谭帝国酒店发起新预订时，向用户请求发送通知的权限。如果用户授予我们权限，就立即显示一条通知，让用户知道，当他的预订状态发生任何变化时，都能收到新状态的通知。

在 my-account.js 中添加下列代码，放在 `addReservation()` 函数定义的上方即可：

```
var showNewReservationNotification = function() {
  navigator.serviceWorker.ready.then(function(registration) {
    registration.showNotification("Reservation Received", {
      body:
        "Thank you for making a reservation with Gotham Imperial Hotel.\n"+
        "You will receive a notification if there are any changes to "+
        "the reservation.",
      icon: "/img/reservation-gih.jpg",
      badge: "/img/icon-hotel.png",
      tag: "new-reservation"
    });
  });
};

var offerNotification = function() {
  if ("Notification" in window &&
      "serviceWorker" in navigator) {
    Notification.requestPermission().then(function(permission){
      if (permission === "granted") {
        showNewReservationNotification();
      }
    });
  }
};
```

我们的代码定义了两个新函数。

`showNewReservationNotification()`

　当用户创建新预订时，显示一条新通知。这个函数会假设用户已经授权应用显示通知。

`offerNotification()`

　确保当前浏览器中支持 service worker 和 Notification API。然后继续请求显示通知的权限；如果权限被授予，就使用 `showNewReservationNotification()` 显示通知。

接下来我们要调用新的函数。依然是在 my-account.js 中，修改 `addReservation` 函数，在创建新预订后，添加一个 `showNewReservationNotification()` 的调用：

```
var addReservation = function(id, arrivalDate, nights, guests) {
  var reservationDetails = {
    id:          id,
    arrivalDate: arrivalDate,
    nights:      nights,
    guests:      guests,
    status:      "Sending"
  };
  addToObjectStore("reservations", reservationDetails);
  renderReservation(reservationDetails);
  if ("serviceWorker" in navigator && "SyncManager" in window) {
    navigator.serviceWorker.ready.then(function(registration) {
      registration.sync.register("sync-reservations");
    });
  } else {
    $.getJSON("/make-reservation", reservationDetails, function(data) {
      updateReservationDisplay(data);
    });
  }
  showNewReservationNotification();
};
```

现在，每当用户发起预订时，`addReservation()` 函数就会向用户请求发送通知的权限（如果尚未授权），然后显示一条新通知（见图 10-7）。

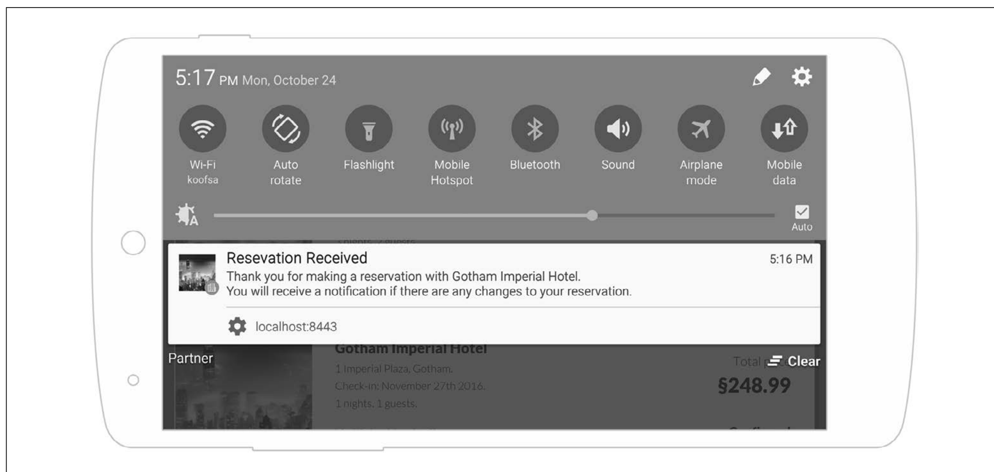


图 10-7：新预订通知

10.3　为用户订阅推送事件

现在我们已经发送了第一条通知，并取得了很大的进展。但是为了能真正让用户受益，我们要在用户离开应用之后向他们发送通知。为此，我们要转而使用 Push API。

让我们再次看看订阅的过程（见图 10-8）。

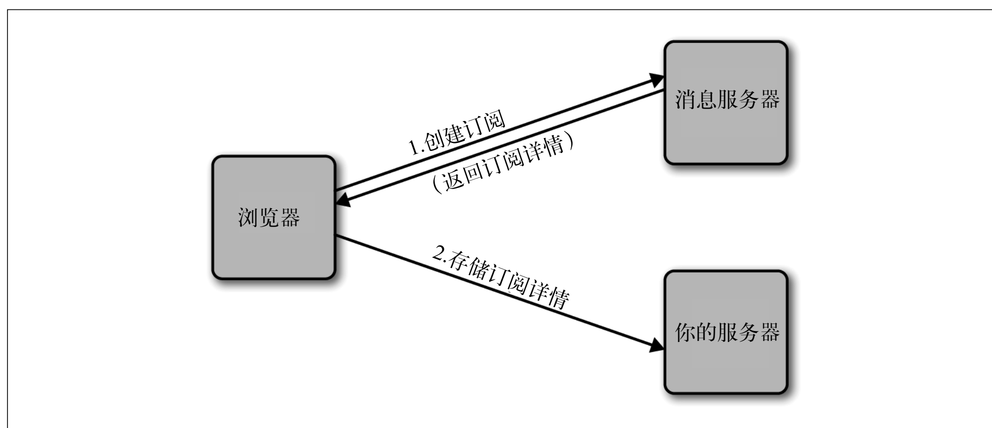


图 10-8：创建并存储推送订阅

首先，我们的脚本要联系消息服务器，要求服务器为用户创建新的订阅。消息服务器随后会将新订阅保存下来，并使用新订阅的详情来响应我们的请求。接下来，我们的脚本要将订阅详情储存到我们的服务器，以便我们可以在随后使用它来发送消息。

在开始创建和存储订阅的过程之前，我们要先花点时间来讨论加密。别担心，这不会花太长时间。

当为用户订阅推送消息时，消息服务器返回的订阅详情对象中，包含了向用户发送无限条消息所需的所有信息。如果任何恶意实体通过服务器获取到订阅详情，或者浏览器中的任何恶意脚本或者插件在用户订阅时读取了详情数据，它们就可以随心所欲地向你的用户发送任何消息了。

为了确保只允许你的服务器发送消息，消息服务器只会接受使用存储在服务器上的秘密私钥签名的消息。要验证消息是否由正确的密钥签名，每个私钥都会有对应的公钥。这个公钥会包含在你的脚本中，在创建新订阅时，公钥会一并发送给消息服务器。随后，公钥会连同订阅详情一起存储在消息服务器中。这个密钥仅会用来验证：从服务器发送给消息服务器的消息，是否由正确的私钥签名了。

你可以把私钥看作是一个只有你的服务器拥有的皇家印章，它可以用来给消息签名，以证明消息是来自于你们皇家的。另一方面，公钥是任何人都可以访问的工具。它不能用来签署消息，只是用来鉴别消息是否确实是通过正确的皇室印章所签署的。

让我们用简单的话语描述一下整个过程：

(1) 创建应用时，生成一个公钥和一个私钥；
(2) 私钥是保密的，保存在服务器中；
(3) 公钥会包含在脚本中，并在创建订阅时被发送到消息服务器；
(4) 消息服务器将公钥连同其他订阅详情一起存储起来；
(5) 当服务器要发送消息时，使用私钥进行签名，随后发送到消息服务器；
(6) 消息服务器使用公钥验证消息是否用正确的私钥签名，如果是，则将消息发送给用户。

看完上述这些步骤，你就可以发现，在开始创建订阅和发送推送消息之前，我们需要生成一个公钥和私钥对。

10.3.1 生成VAPID公钥和私钥

用于签名和验证推送消息的密钥称为 **VAPID 密钥**。VAPID 是“自愿的 Web 推送应用服务器身份证明”（Voluntary Application Server Identification for Web Push），这个创造性的名字和密码学并非密切相关。

为了让事情尽可能简单，我们不会深入研究背后的密码学细节、生成 VAPID 密钥的详情，以及如何为负载签名。相反，我们将使用一个更常用的 Web 推送工具库来隐藏这种复杂性。本书使用了 Node.js 的 web-push 库，但你也可以找到其他许多语言的类似工具库。

首先，我们要在项目中安装 web-push 库。在项目的根目录下，在命令行中运行下列代码，安装 web-push 并将其添加到项目的依赖列表中：

```
npm install web-push --save-dev
```

接下来使用 web-push 来生成一个公钥和一个私钥。

在项目中创建一个名为 generate-keys.js* 文件，并在其中输入下列代码：

```
var webpush = require("web-push");
console.log(
  webpush.generateVAPIDKeys()
);
```

接下来在命令行中执行这个文件：

```
node generate-keys.js
```

这应该能输出一个新的私钥和一个公钥到控制台中：

```
$ node generate-keys.js
{ publicKey: 'yteswBFEx-JuJhyU7XsteR7xOo3nqygyR',
  privateKey: 'IuKbrkM4inNv2MzlzVRDV4YRw4N65N' }
```

你需要把这些密钥保存到安全的地方。

对于哥谭帝国酒店，我选择把私钥和公钥一同保存在 /server 目录的 push-keys.js 文件中。你可能还会注意到，我已经把这个文件添加到项目的 .gitignore 文件中。这意味着当我提交代码时，私钥不会被传到线上。你应该小心保管你的私钥。

为了方便起见，我在 /server 目录中包含了一个名为 generate-push-keys.js 的文件。运行这个脚本时，脚本会为你生成一个新的 push-keys.js 文件，并将新的密钥保存在内。

现在你已经知道如何生成自己的密钥了，可以删除刚才创建的 generate-keys.js 文件，并在命令行中运行下列命令，从而运行 generate-push-keys.js：

```
node server/generate-push-keys.js
```

这条命令会为你创建一个新的密钥对，并保存在 push-keys.js 文件中，如下一节所示。该文件会在稍后被用于向服务端发送消息。

10.3.2 生成GCM密钥

不幸的是，仅仅使用 VAPID 密钥不足以向所有的浏览器发送推送消息。

在 Web 推送协议最终确定下来且 VAPID 达成协议之前，一些浏览器走在了前头，通过非标准的方式实现了推送消息。Chrome 在版本 42 到 51 之间使用了谷歌云消息（Google Cloud Messaging，GCM）来传递推送消息，而且 Opera 和三星浏览器也采用了相同的实现。为了使推送通知也能够在这些浏览器的旧版本上工作，除了 VAPID 密钥之外，你还需要生成 GCM API 密钥。

你可以通过谷歌的 Firebase 云消息接口（过去称为谷歌云消息）获取 GCM API 密钥（又名 FCM API 密钥）。

(1) 从 https://pwabook.com/rebaseconsole 登录 Firebase 控制台。
(2) 使用谷歌账号登录。
(3) 创建新项目。
(4) 在项目页面，点击项目名称旁边的设置图标，进入项目设置页。
(5) 在项目设置中点击"Cloud messaging"。
(6) 你应该能够看到一个项目凭据（Project credentials）区域，其中有一个生成密钥的链接。点击生成密钥，你就能得到属于自己的 GCM 服务端密钥以及发送人 ID（见图 10-9）。

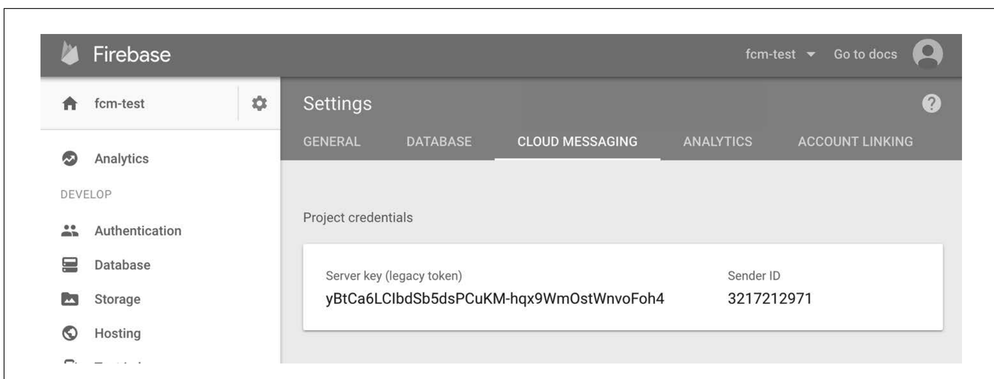


图 10-9：在 Firebase 控制台中生成 GCM 密钥

在 /server 目录中打开 push-keys.js 文件，并将 `GCMAPIKey` 的值设置为你刚才生成的 GCM 服务端密钥值。与此同时，输入服务端管理员的邮箱地址，或者可以联系到你的 URL（这样消息服务器需要联系消息发送者时就有了联系方式）。

修改后的 push-keys.js 文件应该如下所示（但是密钥的值不同）：

```
module.exports = {
  GCMAPIKey: "yBtCa6LClbdSb5dsPCuKM-hqx9WmOstWnvoFoh4",
  subject: "mailto:tal@talater.com",
  publicKey: "yteswBFEX-U7XsteR7x0o3nqygyR",
  privateKey: "IuKbrkM4inNv2MzlzVRDV4YRw4N65N"
};
```

现在服务器知道了 GCM 服务器密钥，是时候将 GCM 发送者的 ID 添加到客户端，以便用来创建新的订阅。

在 /public 目录中编辑网站的 manifest.json 文件，添加一个新的设置项，键名为 `gcm_sender_id`，其值等于 GCM 发送者 ID。

```
{
  "short_name": "Gotham Imperial",
  "name": "Gotham Imperial Hotel",
  "description": "Book your next stay, manage reservations, and explore Gotham",
  "start_url": "/my-account?utm_source=pwa",
  "display": "fullscreen",
  "icons": [
    {
      "src": "/img/app-icon-192.png",
      "type": "image/png",
      "sizes": "192x192"
    },
    {
      "src": "/img/app-icon-512.png",
      "type": "image/png",
      "sizes": "512x512"
    }
  ],
  "theme_color": "#242424",
  "background_color": "#242424",
  "gcm_sender_id": "3217212971"
}
```

现在让我们回到编码当中。

10.3.3 创建新订阅

现在，我们已经把基础打好了，终于可以将注意力转回到浏览器，订阅用户推送消息了。

我们可以使用 service worker 的 `registration` 对象，得到 PushManager 接口。这个接口中包含了一系列的实用方法，包括获取现有的订阅（`getSubscription()`）、检查当前页面是否有权限订阅推送消息（`permissionState()`），最重要的是 `subscribe()` 方法，它可以用来订阅用户推送消息。所有的这些方法都会返回 promise：

```
var subscribeOptions = {
  userVisibleOnly: true
};

navigator.serviceWorker.ready.then(function(registration) {
  return registration.pushManager.subscribe(subscribeOptions);
}).then(function(subscription) {
  console.log(subscription);
});
```

代码一开始定义了一个订阅选项对象，其中只包含了一项设置——`userVisibleOnly`。这个设置项意味着所有推送消息必须对用户可见（即代表你同意为每个推送消息生成通知）。

由于在用户不知情的情况下，在 service worker 中接受消息可能会危及用户隐私，所以目前没有浏览器支持把 userVisibleOnly 设置为 false。如果你试图在创建订阅时，没有将这个值设置为 true，那么消息服务器就会返回错误。

接下来，代码获取了 service worker 的 registration 对象，随后在这个对象上调用了 pushManager 的 subscribe() 方法（传入订阅选项对象）。这个方法会返回一个 promise，promise 完成时，会得到从消息服务器返回的订阅详情对象。

由于这段代码中没有包含 VAPID 密钥，它只能够在支持通过 GCM 发送消息的浏览器中订阅用户，并且是在 manifest.json 文件中引入了 GCM 发送者 ID 的前提下。

让我们来看看，采用 VAPID（如果支持）并且在不支持 VAPID 的情况下回退到 GCM 的做法：

```
var urlBase64ToUint8Array = function(base64String) {
  var padding = "=".repeat((4 - base64String.length % 4) % 4);
  var base64 = (base64String + padding).replace(/\-/g, "+").replace(/_/g, "/");
  var rawData = window.atob(base64);
  var outputArray = new Uint8Array(rawData.length);
  for (var i = 0; i < rawData.length; ++i) {
    outputArray[i] = rawData.charCodeAt(i);
  }
  return outputArray;
};

var subscribeOptions = {
  userVisibleOnly: true,
  applicationServerKey: urlBase64ToUint8Array("yteswBFEX-U7XsteR7x0o3nqygyR")
};

navigator.serviceWorker.ready.then(function(registration) {
  return registration.pushManager.subscribe(subscribeOptions);
}).then(function(subscription) {
  console.log(subscription);
});
```

让我们自下而上看看这段代码。

首先我们修改了订阅选项对象（subscribeOptions），让其接收第二个设置项，名为 applicationServerKey，其中包含了你的 VAPID 公钥（将代码中的随机字符串替换成你的公钥）。不幸的是，pushManager 不能直接接受 VAPID 密钥，我们需要将密钥转换成它能理解的格式。转换的方式取决于代码顶部的 urlBase64ToUint8Array() 函数。这个函数将 VAPID 公钥转换成 pushManager 所需的 Uint8Array 类型。除非你深切地关心密码学，否则不需要深入研究它的工作原理。只需要知道用一个包含你的 VAPID 公钥的字符串来调用它，它就会返回 pushManager 能够理解的数组。

除了我们优雅忽略的 urlBase64ToUint8Array() 复杂性之外，剩余的代码并没有发生太大的变化。唯一的补充就是在设置对象中添加的第二个属性，其中包含了我们的 VAPID 公钥。

就是这样！现在用户已经被订阅了推送消息，你可以在 subscription 变量中获取订阅详情。

此时，你可以使用 Ajax 或者 `fetch` 调用，将订阅对象发送到服务器，以备将来使用。

现在我们了解了如何创建订阅，让我们在应用里实现这一点。

10.3.4 为哥谭帝国酒店用户订阅推送消息

在本章前面，我们为哥谭帝国酒店应用添加了通知的支持——一旦用户发起预订，我们就向他请求发送通知的权限。

现在，让我们来修改那段代码，为授权发送通知的用户同时创建新的推送订阅，并将订阅保存到服务器。

在 my-account.js 中，修改 `offerNotification()` 函数：

```
var offerNotification = function() {
  if ("Notification" in window &&
      "PushManager" in window &&
      "serviceWorker" in navigator) {
    subscribeUserToNotifications();
  }
};
```

我们在 `offerNotification()` 中做了两处修改。首先给 `if` 语句添加了另一个条件，以确保浏览器支持 PushManager。接下来将请求通知权限并为用户订阅推送事件的所有逻辑，提取到 `subscribeUserToNotifications()`，这是我们接下来要编写的另一个新函数。

修改 `addReservation()` 函数的最后一行，让其调用 `offerNotification()` 而不是 `showNewReservationNotification()`。你还可以把 `showNewReservationNotification()` 的代码也删除掉，因为我们不再需要显示那条通知了——取而代之，一旦服务器确认了预订，我们就会推送消息显示通知。

最后，我们在 `offerNotification()` 函数的上面，添加下列代码：

```
var urlBase64ToUint8Array = function(base64String) {
  var padding = "=".repeat((4 - base64String.length % 4) % 4);
  var base64 = (base64String + padding).replace(/\-/g, "+").replace(/_/g, "/");
  var rawData = window.atob(base64);
  var outputArray = new Uint8Array(rawData.length);
  for (var i = 0; i < rawData.length; ++i) {
    outputArray[i] = rawData.charCodeAt(i);
  }
  return outputArray;
};

var subscribeUserToNotifications = function() {
  Notification.requestPermission().then(function(permission){
    if (permission === "granted") {
      var subscribeOptions = {
        userVisibleOnly: true,
        applicationServerKey: urlBase64ToUint8Array(
          "yteswBFEX-U7XsteR7x0o3nqygyR" // 替换为你的公钥
        )
      };
```

```
        navigator.serviceWorker.ready.then(function(registration) {
          return registration.pushManager.subscribe(subscribeOptions);
        }).then(function(subscription) {
          var fetchOptions = {
            method: "post",
            headers: new Headers({
              "Content-Type": "application/json"
            }),
            body: JSON.stringify(subscription)
          };
          return fetch("/add-subscription", fetchOptions);
        });
      }
    });
  };
```

这段代码从我们熟悉的 `urlBase64ToUint8Array()` 函数开始。

接下来，我们定义了 `subscribeUserToNotifications()` 函数。这个函数会请求通知权限，如果成功获取权限，就会创建新的预订并发送给服务器。

它先调用 `Notification.requestPermission()`，向用户请求权限，并返回一个 promise。如果 promise 完成，首先我们要检查权限是否被授予。接下来，我们定义了订阅选项，`applicationServerKey` 设置为 VAPID 公钥，而 `userVisibleOnly` 设置为 `true`。要确保这里使用的是你自己的 VAPID 公钥，可以在 server/push-keys.js 获得。接下来，我们使用 `navigator.serviceWorker.ready` 获取了 service worker 的 `registration` 对象，使用该对象的 `pushManager` 调用 `subscribe()`。当这个 promise 完成后，下一个 `then` 语句块就会运行，此时用户已经授予了权限，并且我们成功向用户订阅了推送消息。

现在我们要做的就是将订阅详情发送到服务器，并保存到数据库中。为此，我们创建一个新的 `fetch` 请求到 `/add-subscription`，设置请求方法为 POST，添加 `Content-Type` 头部并设置值为 `application/json`，让服务器知道我们要传递的是 JSON 数据，最后使用 `JSON.stringify()` 将订阅对象转换成 JSON 字符串。

让我们再看一遍整个过程。

(1) 确保浏览器支持 service worker、Notification API 和 Push API。
(2) 请求显示通知的权限，只有在授权成功的情况下才继续余下操作。
(3) 使用 VAPID 公钥（经过转换后），通过消息服务器创建新的订阅。
(4) 一旦得到订阅详情，就将其发送到服务器并保管起来。

现在我们只剩下一件事：编写服务端代码，将订阅详情保存到服务器的数据库中。这一实现在不同应用之间区别很大，并且跟服务器保存和构造数据的方式有关——但是出发点很简单。通常你可以把订阅详情作为字符串存储到用户表，或者保存到对象存储中。当你需要向用户发送通知时，只需要读取该字符串并且将其转换回对象即可。

你可以在 server/index.js 和 server/subscriptions.js 中看到一个非常简单直接的实现。由于我们的示例应用中没有用户的概念（只服务于单个用户），所以我们简单地把所有订阅都保存在一个 `subscriptions` 对象存储中，没有与任何的用户数据进行联系——在实际应用中，你可不会希望这么做。

10.4 从服务端发送推送事件

我们现在已经准备好从服务端发送消息给用户了。

VAPID私钥和公钥

用于消息签名，以及在支持 VAPID 的浏览器中创建订阅。

GCM API服务端密钥和发送者ID

在不支持 VAPID 的浏览器中作为回退方案，用于消息签名和创建订阅。

订阅详情对象

从消息服务器接收的对象，其中包含了发送消息给特定用户订阅所需的详情信息。

详情中包含了公钥、一个鉴权密钥，以及一个端点——我们要向其发送消息的 URL。

消息

你要发送的消息内容，可以是简单的字符串（例如 `show-new-message-notification`），也可以是包含了更多详情的对象（例如 `{msg: "reservation-confirmation", reservationId: 19, date: "2021-12-19"}`）。

利用所有这些细节，我们就可以构建一个消息服务器的请求来发送这个消息了。事情很快会变得复杂起来，因为这涉及在请求上设置一系列的 HTTP 头部，例如使用 JWT（JSON Web Token）的鉴权头部。

幸运的是，我们可以使用 web-push 库再次绕过这些加密的复杂性，这样就使得发送消息（相对）轻而易举了：

```
var webpush = require("web-push");

var pushKeys = {
  GCMAPIKey: "yBtCa6LClbdSb5dsPCuKM-hqx9WmOstWnvoFoh4",
  subject: "mailto:tal@talater.com",
  publicKey: "yteswBFEX-U7XsteR7x0o3nqygyR",
  privateKey: "IuKbrkM4inNv2MzlzVRDV4YRw4N65N"
};

var subscription = {
  endpoint: "https://fcm.googleapis.com/fcm/send/dQbqPBPWo_A:AHH91bHyhyrG9",
  keys: {
    p256dh: "BEJ_yK1xAC8DFrbXjiRKGVxCh8c8FImUyrNbm8rcVVIvDT3an18ab7011Jw=",
    auth: "o-hRay472334PuqppKq-lg=="
  }
};

var message = "show-notification";

webpush.setGCMAPIKey(pushKeys.GCMAPIKey);
webpush.setVapidDetails(
  pushKeys.subject,
  pushKeys.publicKey,
  pushKeys.privateKey
```

```
);

webpush.sendNotification(subscription, message).then(function() {
  console.log("Message sent");
}).catch(function() {
  console.log("Message failed");
});
```

代码开头引入了 web-push 库。随后我们将前面列出的所有细节引入进来：VAPID 和 GCM 密钥、订阅详情以及我们的消息内容。接下来，可以使用 `webpush.setGCMAPIKey()` 和 `webpush.setVapidDe tails()`，将这些详情配置到 web-push 中。最后，使用 `webpush.sendNotification()` 发送消息，传入订阅对象和消息内容。`webpush.sendNotification()` 会返回 promise，如果消息服务器确认消息可以放入队列等待发送，promise 就会成功，否则如果过程中出现了任何问题，promise 就会失败。

请注意，当消息服务器确定消息可以被发送时，`webpush.sendNotification()` 返回的 promise 才会完成。这并不意味着消息已经成功发送给用户了。用户可能当前处于离线状态，这种情况下，消息服务器会继续尝试发送消息，或者用户甚至可能已经撤销了应用发送通知的权限（这种情况很稀有，但是有可能发生）。

在上一个例子中，我们使用了许多硬编码的值，其中包括了单个订阅的详情，以及简单的文本消息。在现实世界中，示例可能会更加灵活和动态。VAPID 和 GCM 的细节会和业务代码分离开，消息可以被发送到从数据库中检索出的多个订阅，而消息本身也可能包含了更多的细节内容。

我们来看看这在哥谭帝国酒店服务端是如何实现的。

在 subscriptions.js 中，你会看到如下代码：

```
var db = require("./db.js");
var webpush = require("web-push");
var pushKeys = require("./push-keys.js");

var notify = function(pushPayload) {
  pushPayload = JSON.stringify(pushPayload);
  webpush.setGCMAPIKey(pushKeys.GCMAPIKey);
  webpush.setVapidDetails(
    pushKeys.subject,
    pushKeys.publicKey,
    pushKeys.privateKey
  );

  var subscriptions = db.get("subscriptions").value();
  subscriptions.forEach(function(subscription) {
    webpush.sendNotification(subscription, pushPayload).then(function() {
      console.log("Notification sent");
    }).catch(function() {
      console.log("Notification failed");
    });
  });
};
```

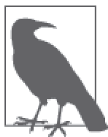
当预订被确认之后，reservations.js 会调用 `notify()` 函数，用来发送消息：

```
subscriptions.notify({
  type: "reservation-confirmation",
  reservation: reservation
});
```

你会看到 subscriptions.js 使用了本地数据库（在 /server/db.js 中定义）和 web-push 库。它还使用了一个称为 push-keys.js 的外部文件，其中存储了发送推送消息所需的密钥（10.3.1 节中介绍了如何生成这个文件）。

你还会注意到，它使用了 `JSON.stringify()` 将接收到的消息转换成字符串。这样我们可以确保传递对象作为消息是可行的，如上述的示例代码所示。

最后，从数据库中获取订阅详情对象，而 `forEach()` 循环可以保证消息被发送给所有人。在你的应用中，更可能的做法是一次只向一个用户发送消息，或者为每个用户定制每条消息的内容。为了保持代码简单，我们的示例服务器只知道如何处理单个用户，因此每次确认预订通知都会发送给所有订阅者。



本章介绍了服务端代码，这是本书中的第一次。我一直保持这份代码尽可能简单，以便突出核心概念。我们的实现使用了 Node.js 和 web-push 库来处理发送推送消息。你可以找到其他编程语言中许多类似的库。

如果你正在使用哥谭帝国酒店进行编程练习，不需要在服务端实现任何新的内容。上述所有服务端代码都已经在你下载的源代码中实现了。

10.5　监听推送事件并显示通知

现在，我们的前端代码知道如何获取权限来向用户显示通知、创建订阅并存储在服务器中，服务器知道如何在确认预订时向用户的浏览器发送推送消息。

接下来，我们将注意力转回浏览器，看看 service worker 是如何监听这些消息并进行操作的。

正如我们看到的那样，Push API 和 Notification API 要求的是同一个权限。这意味着一旦 service worker 收到推送消息，我们就可以显示通知了，代码可以像下例这样简单：

```
self.addEventListener("push", function() {
  self.registration.showNotification("Push message received");
});
```

当推送消息到达浏览器时，会在 service worker 中触发 push 事件。即使用户在几周之内都没有访问过我们的网站，service worker 依然会在消息到达时立刻开始行动；通过通知，我们的应用就有机会重新召回用户（见图 10-10）。

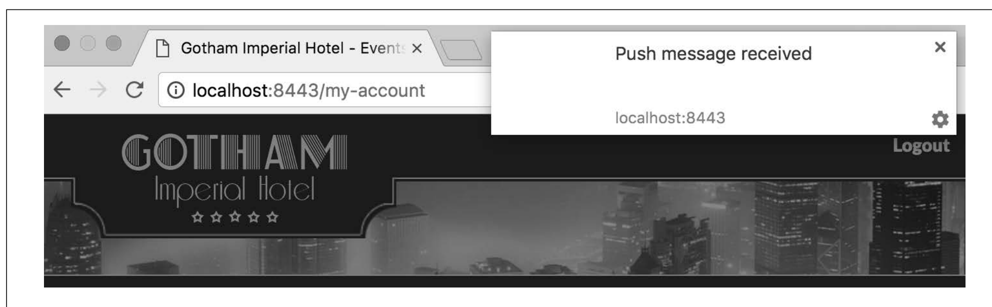


图 10-10：响应推送事件时显示的通知

在 service worker 中显示通知的代码和我们在 10.2 节中看到的代码相同。唯一的区别是，在 service worker 中，可以轻松使用 `self.registration` 访问 `registration` 对象。

在 `push` 事件监听器中，你可以通过 `PushEvent` 对象的 `data` 属性（传递给事件监听器的第一个参数），访问推送消息的内容：

```
self.addEventListener("push", function(event) {
  var message = event.data.text();
  self.registration.showNotification("Push message received", {
    body: message
  });
});
```

如图 10-11 所示，`PushEvent` 对象属性中包含了 `text()` 方法，以简单的字符串形式返回消息内容。此外它还有 `json()` 方法，可以将消息内容解析为 JSON，并以对象形式返回（见图 10-12）。

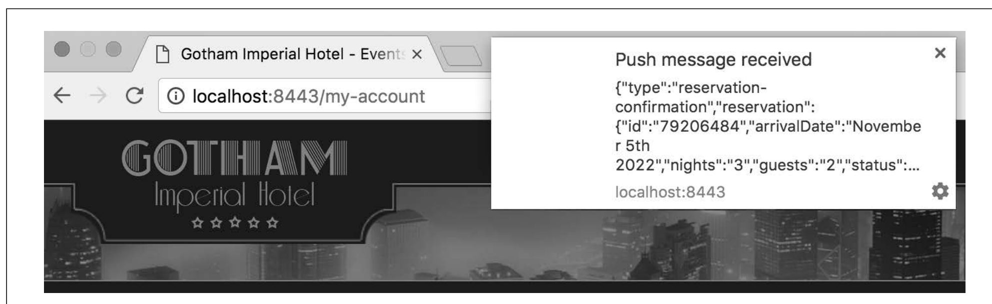


图 10-11：在 push 事件发生时，通过 event.data.text() 显示通知

```
self.addEventListener("push", function(event) {
  var message = event.data.json();
  self.registration.showNotification("Push message received", {
    body: "Reservation for "+message.reservation.arrivalDate+" has been confirmed."
  });
});
```

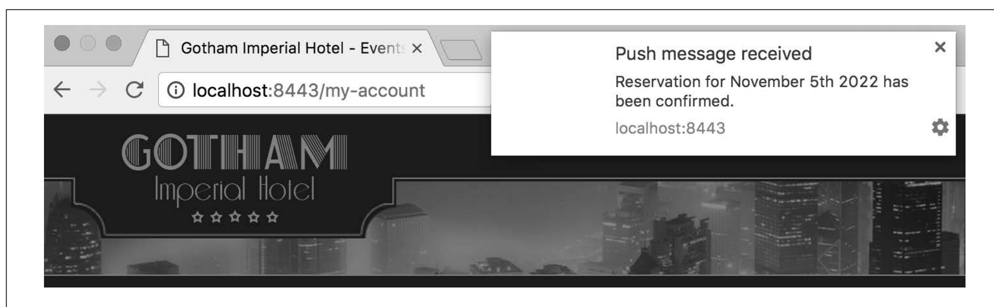


图 10-12：在 push 事件发生时，通过 event.data.json() 显示通知

让我们结合目前为止所学的内容，在用户预约被确认时，为哥谭帝国酒店的用户显示一条很炫的通知。

在 serviceworker.js 中，在文件的末尾添加下列代码：

```
self.addEventListener("push", function(event) {
  var data = event.data.json();
  if (data.type === "reservation-confirmation") {
    var reservation = data.reservation;
    event.waitUntil(
      updateInObjectStore(
        "reservations",
        reservation.id,
        reservation)
      .then(function() {
        return self.registration.showNotification("Reservation Confirmed", {
          body:
            "Reservation for "+reservation.arrivalDate+" has been confirmed.",
          icon: "/img/reservation-gih.jpg",
          badge: "/img/icon-hotel.png",
          tag: "reservation-confirmation-"+reservation.id,
          actions: [
            {
              action: "details",
              title: "Show reservations",
              icon: "/img/icon-cal.png"
            }, {
              action: "confirm",
              title: "OK",
              icon: "/img/icon-confirm.png"
            },
          ],
          vibrate:
            [500,110,500,110,450,110,200,110,170,40,450,110,200,110,170,40,500]
        });
      })
    );
  }
});
```

我们的新事件监听器会耐心等待推送事件，当这样的推送消息到达 service worker 后，事件监听器会检索 `PushEvent` 中包含的数据，并根据其包含的 `type` 属性决定如何进行操作。如果 `type` 的值等于 `reservation-confirmation`，代码就会知道，它需要使用 `updateInObjectStore()` 更新 IndexedDB 中的预订数据，并使用 `self.registration.showNotification()` 显示通知（见图 10-13）。

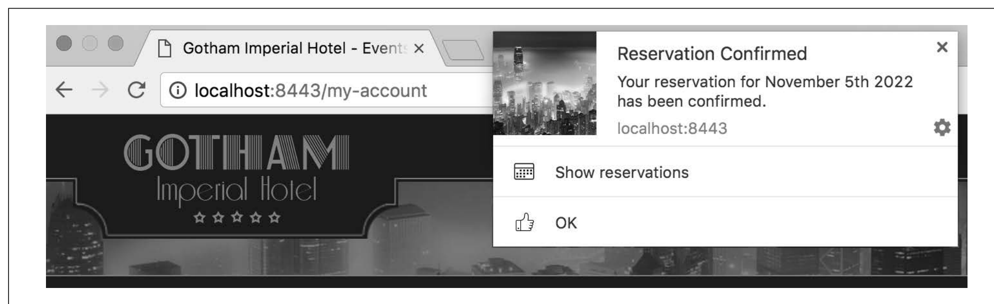


图 10-13：预订确认通知的最终效果



快速提醒：哥谭帝国酒店服务器发送的消息的结构如下：

```
{
  "type": "reservation-confirmation",
  "reservation": {
    "id": "79212418",
    "arrivalDate": "November 5th 2022",
    "nights": "3",
    "guests": "2",
    "status": "Confirmed",
    "bookedOn": "2016-10-31T15:40:41+02:00",
    "price": 636
  }
}
```

将推送消息的数据构建为包含 type 和预订详情的对象，完全是主观的选择。我们也可以将其构造成不包含 type 的对象。甚至可以将整条消息生成一个包含最终通知文本的字符串，或者是诸如 `reservation-confirmation,79212418` 这样的字符串，随后在 service worker 中进行解析，并根据 ID 从 IndexedDB 中获取预订详情。

让我们更加仔细地查看新事件监听器的代码。

首先，你可能会注意到，push 事件监听器代码使用了 `event.waitUntil()`，以确保在原始 push 事件完成之前，先要等待更新 IndexedDB 和通知的代码一起完成。正如我们在第 3 章看到的那样，`waitUntil()` 会延长事件的生命周期，直到传递进去的 promise 完成为止。在这个例子里，我们传入的是 `updateInObjectStore()`，它会返回一个 promise，随后将这个 promise 传递给 `showNotification()`，后者也会返回一个 promise。

如果我们没有告诉 `PushEvent` 等待其中的代码执行完成，可能会发现启动了需要花费

更长时间才能完成的操作（例如发起网络请求），但是由于浏览器可能认为操作完成时 PushEvent 已经结束，所以 service worker 可能就不能对结果进行操作了。

在开始显示通知的逻辑之前，代码首先调用了 updateInObjectStore()，并更新了 IndexedDB 中的预订详情。通过在 push 事件中进行这一操作，我们就可以确保本地的预订数据可以一直保持最新。如果用户接收到推送通知，告知他其中一个预订已经确认，那么他在离线访问应用时，最新的预订数据（包括已确认的预订）就可以展示出来。

接下来代码调用了 showNotification()。这里使用的语法我们应该很熟悉了，但你可能还会注意到，除了自定义消息、花哨的徽章标记和图标、vibrate 选项播放的主题曲之外，通知还包含了两个按钮。它们是使用通知选项对象的 actions 属性来创建的。

每个通知操作都是由 title（按钮文本）、icon（在文本旁边显示的图标）以及 action（用于表示该操作的名称）组成的。显然，它还遗漏了一些东西：实际进行某项操作的方式。

由于通知是在浏览器之外（操作系统层面）渲染的 UI 元素，用户可能会在通知创建几个小时之后才进行操作（例如，在午夜弹出的通知），因此设置回调或者 promise 来等待操作是没有意义的。相反，可以将通知上进行的操作作为独立的事件发送到 service worker。通过监听这些事件，我们就可以基于用户交互（忽略通知，或者是点击两个按钮之一）进行对应的操作。

编辑 serviceworker.js，将下列代码添加到文件结尾：

```
self.addEventListener("notificationclick", function(event) {
  event.notification.close();
  if (event.action === "details") {
    event.waitUntil(
      self.clients.matchAll().then(function(activeClients) {
        if (activeClients.length> 0)  {
          activeClients[0].navigate("http://localhost:8443/my-account");
        } else {
          self.clients.openWindow("http://localhost:8443/my-account");
        }
      })
    );
  }
});
```

这段代码会监听 notificationclick 事件。每当应用创建的任何通知被用户点击时，就会发送这些事件。

我们的事件监听器在一开始会调用 event.notification.close() 来关闭通知。一旦用户和通知进行了交互，我们就不需要再保留通知了。这也确保了不同设备、操作系统和浏览器上的体验都一致——其中一部分会在用户点击时自动关闭通知，另一部分则只会在你发出指令后才会关闭。

接下来，我们需要了解用户如何与通知进行交互。由于我们的网站目前只有一个通知，并且我们只关心当“Show reservations”按钮被点击时会发生什么，所以我们检查了事件的 actions 属性。action 属性会包含被点击的操作名称（如果没有点击任何一个操作，值就是空字符串）。属性值等于我们指定的 action 属性（我们命名为 details 和 confirm）。如

果用户点击的操作是 `details`，我们就要跳转到应用的 My Account 页面。此时，我们可以简单地调用 `self.clients.openWindow(url)` 来打开一个新窗口。但是为了提供更优的用户体验，我们可以首先检查应用中是否有激活的窗口；如果有，则在那个窗口上跳转到 My Account 页面。

如果你对于检查打开窗口（`self.clients.matchAll`）的代码感到陌生，请参考第 8 章。

询问通知

上述的用例非常简单。我们的网站里只有一种通知类型（确认预订），因此我们并不在乎用户点击了哪种类型的通知。在更加复杂的例子中，我们可能会同时打开多个通知来通知新事件，以及同时显示多个确认预订的通知。

如果我们想要知道哪种类型的通知触发了 `notificationclick` 事件（例如：是新事件还是确认预订），用户点击的是哪个特定的通知（例如：用户点击的是万圣节派对通知的回复按钮，还是新年舞会的通知），该怎么办？

有几种方法可以确定用户与之交互的是哪一个通知。

最简单的方法就是我们刚才所看到的那样。只需要检查用户点击的操作名称即可。在我们的场景下这样足够了，因为我们不关心通知的类型（我们只有一种类型）以及通知所提及的是哪一个预订。

另一种方法是，读取通知窗口的名称。这个名称就是我们之前创建通知的时候分配的 `tag` 标签。以下就是这种方法，可以根据通知的 `tag` 决定如何处理：

```
self.addEventListener("notificationclick", function(event) {
  if (event.notification.tag === "event-announcement") {
    self.clients.openWindow("http://localhost:8443/events");
  } else if (event.notification.tag === "confirmation") {
    self.clients.openWindow("http://localhost:8443/my-account");
  }
});
```

当用户点击的通知标签是 `event-announcement` 时，我们采取第一种操作；如果通知标签是 `confirmation`，则代码采取另一种操作。

第三种方法是给每一条通知传递数据：

```
self.addEventListener("push", function(event) {
  var data = event.data.json();
  var reservation = data.reservation;
  self.registration.showNotification("Reservation Confirmed", {
    tag: "reservation-confirmation",
    data: reservation
  });
});

self.addEventListener("notificationclick", function(event) {
  event.notification.close();
  if (event.notification.tag === "reservation-confirmation") {
```

```
        var reservation = event.notification.data;
        self.registration.showNotification("Notification clicked", {
          body:
          "Notification tag: "+event.notification.tag+"\n"+
          "Notification reservation date: "+reservation.arrivalDate
        });
      }
    });
```

这个代码示例演示了当我们在推送事件中创建通知时，可以将 `data` 属性设置为包含预订的详情。随后当通知被点击时，我们可以通过 `event.notification.data` 访问数据，并使用它来展示第二条通知（或者在网站中打开特定的页面，指向特定的预订）（见图 10-14）。

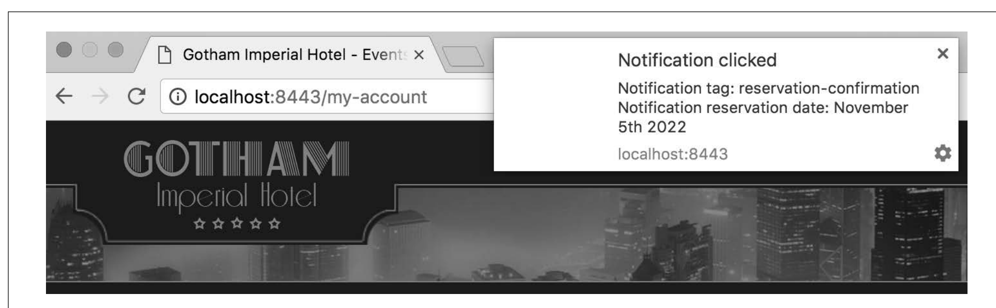


图 10-14：响应 `notificationclick` 事件时显示的通知

10.6 小结

本章中对哥谭帝国酒店 Web 应用做的改进，真正将渐进式 Web 应用结合到了一起。

我们不仅可以在任何连接状态下给予用户最佳的体验，还可以让用户在离开应用后保持状态更新。

让用户知悉预订的更新、在用户到达之前发出提醒，甚至在用户在哥谭逗留期间提供建议，这些能力使得我们可以显著地提升用户的体验。

第11章

渐进式Web应用的用户体验

11.1 优雅与信任

渐进式 Web 应用呈现了 Web 工作方式的一种真正的范式转换。它们提供了超出用户期望的 Web 体验。

用户并不期待 Web 应用离线的时候能够继续工作。然而，渐进式 Web 应用超出了用户的期望。

用户并不期待在与用户相关的新信息出现时，Web 应用会向他们发送更新。然而，渐进式 Web 应用再次超出了用户的期望。

用户并不期待全屏幕的体验、从主屏幕启动、外观和行为宛如原生应用。然而，渐进式 Web 应用又超出了用户的期望。

另一方面，当用户访问 Web 应用时，会期待加载的内容是最新的。但是，如果用户没有意识到自己处于离线状态，他可能同样没有意识到，屏幕上显示的内容可能是几小时前，甚至是几天前的。对用户来说，渐进式 Web 应用似乎没有交付用户所期望的体验。

虽然 Web 和原生应用之间的能力差距正在迅速缩小，但是渐进式 Web 应用的能力与用户的期望与感知之间，仍然存在着很大的差距。

从长远来看，随着越来越多用户使用渐进式 Web 应用，期望与现实之间的差距将会缩小和消失。但是在这发生之前，用户期望和渐进式 Web 应用之间的这种不协调，给我们带来了新的挑战。我们可以通过与用户进行适当的交流，来解决这个问题。



作为开发人员，我们的角色不是教育用户有关 Web 的知识，而是要引导并帮助用户在应用中实现他们的目标。我们可以通过清晰的沟通（包括口头上和视觉上）来帮助用户理解我们的应用，以此来实现这一点。随着时间推移，当用户和开发者在渐进式 Web 应用上花费越来越多的时间，就会出现通用的模式。

想想看，在短短几年内，汉堡包图标 ≡ 几乎成了移动端导航菜单的代名词。在不远的将来，离线也会有属于自己的汉堡包时刻。

在许多方面，随着渐进式 Web 应用在能力上赶超了原生应用，原生应用相比 Web 的优势归结为了信任问题。用户无论在哪里都会信任他们的原生应用，即使在飞机上运行消息应用之前，也不需要三思而后行。然而，用户却不会在起飞后打开浏览器。用户信任原生应用会使用通知来保持状态更新，但是在访问网站时，却要通过一遍又一遍地刷新网站来检查更新。

随着 Web 和原生应用之间的差异变得主要是信任问题，在 Web 应用中传达一种信任感就变得越发重要。当用户在使用应用时丢失了连接，可以通过与用户沟通来增强用户的信任，让他相信其工作不会丢失。当应用被完全缓存时，不妨让用户知道，在离线时可以使用你的应用。在请求权限以发送推送消息之前，让用户确切地知道他可以从通知中获益，以及通知可能包括哪些内容。要记住，信任不仅仅是关于你的渐进式 Web 应用的能力，还包括让用户知道你不会滥用这些能力。

本章会探讨与用户沟通、将信息传达给用户的不同沟通模式。我们还会看到一些增强渐进式 Web 应用界面，从而改善用户体验并提高网站成功概率的大好机会。我们在第 5 章中探索了构建离线优先应用的方式，并优雅地处理了连接的变化，上述这些概念将与第 5 章的内容紧密联系在一起。一旦应用优雅且成功地处理了连接的变化，并与用户清晰地进行交流，灌输信任感，那么应用的体验就可以真正与任何原生应用相媲美了。

11.2 从service worker传递状态

让我们从可能希望与用户沟通的一种信息开始，看看如何在应用中实现它。

我们添加到哥谭帝国酒店应用的第一个增强是让它可在用户离线的情况下也能正常工作。通过让应用在任何连接下无缝地工作，我们的确改善了用户的体验。但是，如果用户不知道自己离线，并且没有意识到自己查看的内容可能是过时的呢？我们可以通过检测用户离线并正在查看缓存内容，向用户展示消息，让他知道自己查看的内容可能过时了。

我们可以通过两步来完成：

(1) 在 service worker 中，检测用户离线并在查看缓存内容，并将此传达给页面；
(2) 在页面中监听消息的传达，并通知用户。

在我们的应用中，大多数动态内容要么从 IndexedDB 返回，要么从缓存回退到网络请求。每次真正从网络请求的唯一内容（仅在用户离线时，回退到缓存响应）只有事件数据。因此这个请求就是检测用户离线并与页面通信的好机会。



这个请求还适合用来检测何时将信息传达给用户，因为事件数据只有在页面的 DOMContentLoaded 事件触发后才会加载。如果我们试图在页面本身（HTML）的请求上进行通信，页面可能没有准备好接收我们的消息并显示通知。在这种情况下，我们就必须要修改 service worker 的代码，在发送消息之前等待页面加载完成。

通过在命令行运行下列命令，确保代码处于上一章结束时的状态：

```
git reset --hard
git checkout ch11-start
```

我们首先修改 serviceworker.js，修改 fetch 事件监听器中处理 /events.json 的部分：

```
} else if (requestURL.pathname === "/events.json") {
  event.respondWith(
    caches.open(CACHE_NAME).then(function(cache) {
      return fetch(event.request).then(function(networkResponse) {
        cache.put(event.request, networkResponse.clone());
        return networkResponse;
      }).catch(function() {
        self.clients.get(event.clientId).then(function(client) {
          client.postMessage("events-returned-from-cache");
        });
        return caches.match(event.request);
      });
    })
  );
}
```

这段代码中的大部分已经在 5.5 节中做出了解释。唯一的补充是 `catch` 语句块中的前三行。当 events.json 网络请求失败时，`catch` 语句块就会执行，代码会发送一条消息到客户端，告知请求了事件文件。消息的内容（数据）是我为这种事件类型选择的一个简单字符串——`events-returned-from-cache`。

接下来，我们需要确保页面监听了 `message` 事件，并显示通知。我们在 app.js 中添加下列事件监听器：

```
if ("serviceWorker" in navigator) {
  navigator.serviceWorker.addEventListener("message", function (event) {
    if (event.data === "events-returned-from-cache") {
      alert(
        "You are currently offline. The content of this page may be out of date"
      );
    }
 });
}
```

这段代码首先确保了用户浏览器支持 service worker。接下来，它将添加一个新的事件监听器，监听 `message` 事件。当检测到这个事件时，检查该消息的内容（在 `event.data` 中可以获取），如果能与我们选择的事件名称相匹配，则向用户显示警报。有关如何在 service worker 和页面之间发送消息的内容请参见第 8 章。

显示 `alert` 给用户，显然不是我们想要的流畅用户体验。让我们来改进一下。

11.3 使用Progressive UI KITT通信

在实现剩余的消息之前，让我们来看一个简便的库，它可以使我们和用户的交流更加容易。

Progressive UI KITT 是一个小型库，可以处理 service worker 和页面之间的通信，并且可以向用户渲染通知。它可以处理发送通知给一个或者多个窗口的情况，可以在通知中引入按钮，并且可以轻松定制任何视觉样式与视觉主题。

让我们来看看如何使用 Progressive UI KITT 添加与上一节中相同的离线消息（见图 11-1）。

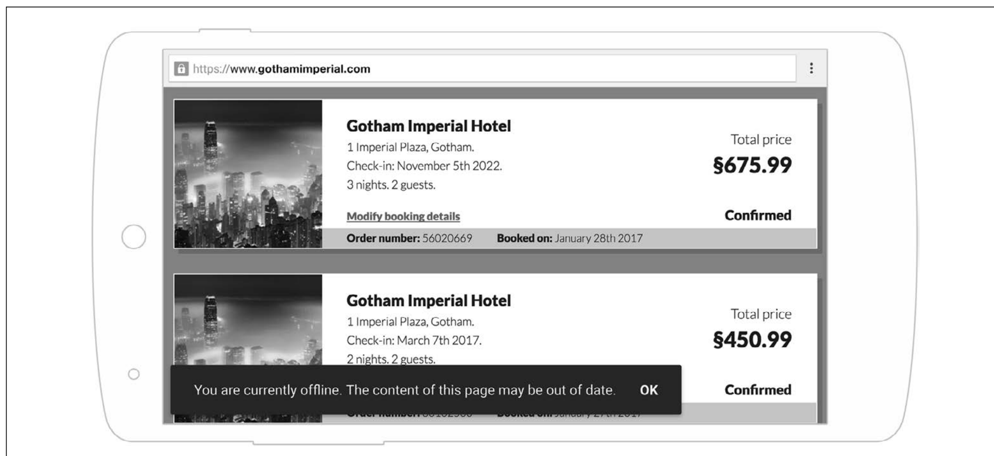


图 11-1：Progressive UI KITT 的离线消息



如果你已经在代码中实现了上一节中的两处修改，现在可以回滚那些修改。我们的新代码会替换那些代码。

Progressive UI KITT 可以在项目的 public/js/vendor/progressive-ui-kitt 目录中获取。要使用它，我们需要三个文件。

- progressive-ui-kitt.js——主要的库文件。在任何需要显示通知的页面中都要引入。
- themes/flat.css——用来调整通知样式的主题文件。可以自由地替换成 themes 目录中的任何其他文件，或者创建你自己的主题文件。
- progressive-ui-kitt-sw-helper.js——包含了要引入到 service worker 中的辅助函数，可以在 service worker 中用来触发任何页面的通知。

在开始前，我们要确保 Progressive UI KITT 及其样式表都能够缓存在 service worker 中，以便我们可以在用户离线时使用它们。

在 serviceworker.js 文件中，添加下列文件到 `CACHED_URLS` 数组中：

```
"/js/vendor/progressive-ui-kitt/themes/flat.css",
"/js/vendor/progressive-ui-kitt/progressive-ui-kitt.js"
```

接下来在 serviceworker.js 的顶部，添加下面这行代码，引入 Progressive UI KITT 的 service worker 辅助方法：

```
importScripts("/js/vendor/progressive-ui-kitt/progressive-ui-kitt-sw-helper.js");
```

最后需要在任何显示通知的页面中引入和初始化 Progressive UI KITT。

在 index.html 和 my-account.html 的代码底部，添加下列代码，放在闭合标签 `</body>` 之前：

```
<script src="/js/vendor/progressive-ui-kitt/progressive-ui-kitt.js"></script>
<script>
ProgressiveKITT.setStylesheet("/js/vendor/progressive-ui-kitt/themes/flat.css");
ProgressiveKITT.render();
</script>
```

这段代码包含了 Progressive UI KITT 的主文件。选择了通知的样式（这里使用的是扁平化主题），并通过调用 `render()` 初始化 KITT。

现在 KITT 已经在页面和 service worker 中准备就绪了，我们可以创建第一条消息。

在 serviceworker.js 中，修改 fetch 事件监听器中处理 /events.json 的部分：

```
} else if (requestURL.pathname === "/events.json") {
  event.respondWith(
    caches.open(CACHE_NAME).then(function(cache) {
      return fetch(event.request).then(function(networkResponse) {
        cache.put(event.request, networkResponse.clone());
        return networkResponse;
      }).catch(function() {
        ProgressiveKITT.addAlert(
          "You are currently offline."+
          "The content of this page may be out of date."
        );
        return caches.match(event.request);
      });
    })
  );
}
```

我们唯一添加的代码，是修改了 `catch` 代码块，调用了 `ProgressiveKITT.addAlert`，并传入我们的消息内容。

就这么简单。只需要一条简单的命令，KITT 就会完成剩下的工作。下次用户在离线访问我们的应用时，就能看到一条通知，如图 11-1 所示。



你可以使用 KITT 创建普通消息、弹窗和确认消息，分别对应的调用是 `ProgressiveKITT.addMessage()`、`ProgressiveKITT.addAlert()` 和 `ProgressiveKITT.addConfirm()`。

弹窗中包含文本和单个按钮的写法（默认按钮标签是 OK）：

```
ProgressiveKITT.addAlert("Caching complete!");
ProgressiveKITT.addAlert("Caching complete!", "Great");
```

确认消息包含文本和两个按钮（默认标签是 OK 和 Cancel）：

```
ProgressiveKITT.addConfirm("Caching complete!");
ProgressiveKITT.addConfirm("Caching complete!", "Great", "OK");
```

普通消息只包含文本，没有包含其他内容：

```
ProgressiveKITT.addMessage("Caching complete!");
```

由于普通消息中没有包含任何按钮，你可能要连同 `hideAfter` 选项一起使用，以在一段时间之后自动隐藏消息：

```
ProgressiveKITT.addMessage("Expiring message", {hideAfter:2000});
```

所有这些命令都可以在 service worker 和页面中正常运行。

要了解 KITT 接受的完整参数列表、如何将回调函数添加到按钮，以及完整的文档，请参阅 https://pwabook.com/kitt。

11.4　渐进式Web应用中的常见消息

除了上一节中显示的离线消息之外，还有哪些其他信息需要传达给用户呢？这个问题的答案取决于你的应用，但是这里可以提供一些思路。

11.4.1　缓存完成

一旦 service worker 完成安装，并缓存了显示应用所需的所有静态资源，你可能想要给用户显示一个消息，让他知道网站现在可以离线工作了：

```
self.addEventListener("install", function(event) {
  event.waitUntil(
    caches.open(CACHE_NAME).then(function(cache) {
      return cache.addAll(CACHED_URLS).then(function() {
        ProgressiveKITT.addMessage(
          "Caching complete! Future visits will work offline.",
          {hideAfter: 2000}
        );
        return Promise.resolve();
      });
    })
  );
});
```

11.4.2　页面已缓存

在 8.1 节中，我们看到一个提供旅游指南的网站，其中列出了哥谭市的每一家餐馆。由于哥谭有数以千计的餐馆，我们认为将所有餐馆的详情都缓存起来是不合理的。取而代之的是，我们选择只缓存用户感兴趣的餐馆（用户访问过的页面）。如何将这件事传达给用户，让用户知道，即使在离线情况下也能打开这个新餐馆的页面呢？

```
self.addEventListener("message", function(event) {
  if (event.data === "cache-current-page") {
    var sourceUrl = event.source.url;
```

```
      caches.open("my-cache").then(function(cache) {
        return cache.addAll([sourceUrl]).then(function() {
          ProgressiveKITT.addMessage(
            "This restaurant's details can now be accessed offline.",
            { hideAfter: 2000 }
          );
          return Promise.resolve();
        });
      });
    }
  });
```

在 11.7 节中，我们还将看到传达这种信息的一种视觉方式。

11.4.3 操作失败，但会在用户恢复连接时完成

在第 7 章中，我们学习了如何使用后台同步，确保用户所采取的任何操作都能可靠地完成，即使是在连接失败的情况下。但是，当用户在移动设备上采取重要操作或者填写表单时，如果连接失败，用户可能会很激动。在这种情况下，我们可以向用户保证，操作将会在恢复连接之后立即进行：

```
self.addEventListener("sync", function(event) {
  event.waitUntil(
    saveChanges().catch(function() {
      ProgressiveKITT.addAlert(
        "You are currently offline, but your reservation has been saved."
      );
    })
  );
});
```

11.4.4 启用通知

一旦用户订阅了推送通知，你就可以告知用户，在更新可用时他会收到通知：

```
navigator.serviceWorker.ready.then(function(registration) {
  return registration.pushManager.subscribe(subscribeOptions);
}).then(function() {
  ProgressiveKITT.addMessage(
    "Thank you. You will be notified of any changes to your reservation.",
    {hideAfter: 3000}
  );
});
```

比起立即滥用通知权限（发送一条关于通知的通知）来告知用户他会收到通知，这种显示消息的方式更加巧妙。

11.5 选择正确的用词

和用户沟通并增强用户对应用的信任感，选择正确的用词是其中一个重要的部分。

如果用户花了几分钟时间认真地编辑图片、添加了艺术滤镜和井号话题描述，只是在点击提交按钮前的一秒丢失了连接，请不要使用不详的“网络错误”信息进行响应。相反，要让用户放心，他的图片、消息和辛苦工作不会丢失，并且可以在随后的时间再次发送。

你甚至可以根据连接状态的变化，修改应用界面的措辞。如果你将保存按钮的文字改成“保存到本地”，或者是将发送按钮改成“当在线时发送”，就可以提升用户对应用的信心——相信他们的工作不会丢失。否则，他们就会盯着按钮，迟疑如果点击之后会发生什么。

11.6 不要直奔主题

消息和信任至关重要的另一个场景，是当请求用户准许我们做某事情时。

在第 10 章中，我们向用户请求了推送通知的权限，以此作为通知和重新召回用户的一种方式。但是请求用户授权时的用户体验是缺失的。我们只是简单地打开了授权对话框，但是没有为用户提供任何上下文，也没有解释通知的使用方式。

不幸的是，我们经常看到这种用户体验做得很差，并且会导致自毁的结果，值得我们花时间思考一种更好的选择。

在考虑请求发送推送通知的权限之前，先要考虑两件事情：**时机**和**消息**。

记住，一旦用户拒绝了权限请求，你就不能再请求了。出于这个原因，考虑请求的**时机**是至关重要的。如果用户刚到达你的网站，你就打开授权对话框，那么用户肯定会立即拒绝请求，可能还会离开你的网站。相反，应在通知为用户提供了有形利益时，再请求权限。例如，如果你打算发送一条通知，提醒用户关于预订的变更，那么在用户发起预订之后再请求权限。如果你负责的是新闻网站，请考虑在用户表现出兴趣之后，再发送关于足球新文章的通知。要记住，你只有一次请求机会——如果你在用户看到价值之前就发出请求，可能会永远失去这个机会。

接下来要考虑**消息**。当你打开浏览器原生的授权对话框时，你是不能控制消息内容的，呈现给用户的是 `<URL> wants to send you notifications`（见图 11-2）。

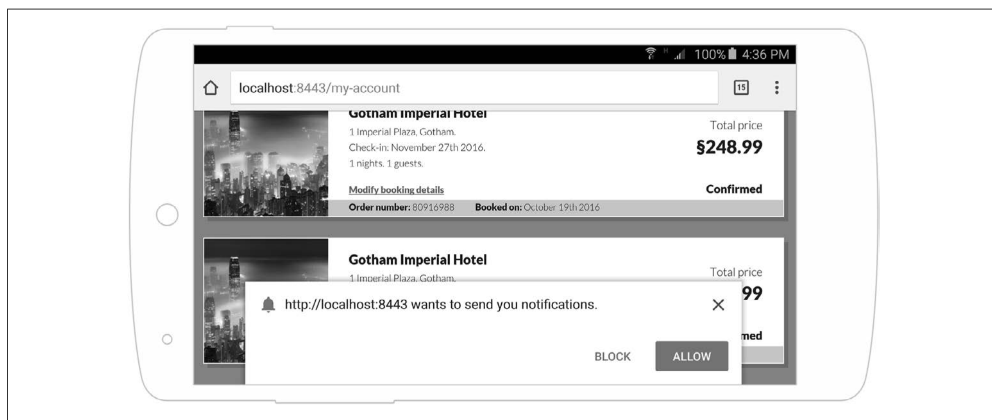


图 11-2：Chrome 的默认通知授权对话框

从用户的角度来看这条消息，它给用户带来的利益是尚不清楚的。有些网站想要给我发送通知，但是里面包含了什么呢？是哪种类型的通知？点击 Allow 会导致我被垃圾信息骚扰吗？它没有给用户灌输信心，而是引起了用户的疑问与质疑。

这是一种糟糕的用户体验，我们应该树立更高的目标。

与其立即触发原生的授权请求，不如考虑创建一套属于你自己的界面，让用户自己去触发。在这套界面里，你可以控制消息内容——让用户清楚通知是什么，以及他会收到哪种类型的消息。随后用户可以选择启用通知，在这种情况下，你就可以调用 `Notification.requestPermission()`；用户也可以拒绝启用通知。是的，这确实增添了一个额外的步骤，在授权时需要用户点击两次，但是这种做法几乎总是能够带来更高的转化率。



创建属于你自己的界面来提供通知，有两个好处。

(1) 如果用户拒绝了你的建议，你可以在将来再次尝试提供建议。如果用户拒绝了浏览器的 `requestPermission()` 对话框，你就不能再次请求了。

(2) 随后，你可以复用这个订阅通知的界面，供用户取消订阅。

说到**消息**，销售人员有句老话说得好：不要试图推销产品的功能，而要推销它的好处。这不是要你使用一些虚伪的营销技巧，而是说要确保用户能够理解你在提供什么，而用户需要同意什么。

在编写消息时，不要只提供功能（例如“开启推送消息”），而要描述用户会获得的好处（例如“在订单发货时收到通知”）。

你认为图 11-3 所示的两个消息中，哪一个能够导致更高的转换率？

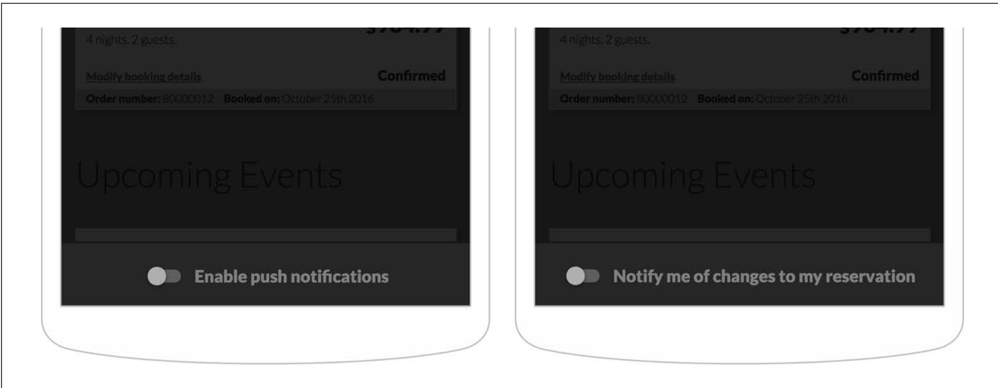


图 11-3：两种不同的通知方式

作为用户，右图中消息带给我的好处是显而易见的。消息对我而言非常重要，我宁愿额外点击 17 次（如果必须的话），只要不错过我的预订发生的变化。

让我们看看如何提升哥谭帝国酒店通知的用户体验。

在 my-account.js 中，将 `offerNotification()` 函数修改为以下内容：

```
var offerNotification = function() {
  if ("Notification" in window &&
      "PushManager" in window &&
      "serviceWorker" in navigator) {
    if (Notification.permission !== "granted") {
      showNotificationOffer();
    } else {
      subscribeUserToNotifications();
    }
  }
};
```

旧的代码总是会调用 `subscribeUserToNotifications()`（在必要时请求授权，随后在必要时创建订阅），现在我们只会在用户已经授予通知权限时才调用它。否则，我们不希望立即触发原生的授权对话框，而是使用我们自己的对话框。我们调用 `showNotificationOffer()`，这个方法会显示一个 div（`#offer-notification`），其中包含了一个打开通知的链接。

在 my-account.js 的底部添加下列代码，确保在点击 div 内的通知链接时，`subscribeUserToNotifications()` 会被调用：

```
$("#offer-notification a").click(function(event) {
  event.preventDefault();
  hideNotificationOffer();
  subscribeUserToNotifications();
});
```

我们所做的两次改动都很小。如果用户尚未授权，我们就不再在用户发起预订时调用 `subscribeUserToNotifications()`。相反，我们显示自己的提供通知的界面元素，并且只在用户选择同意时，才调用 `subscribeUserToNotifications()`。

通过这种方式，我们就能控制消息的**时机**，只有在知道给用户提供了真正的好处后才显示出来。同时我们也控制了**消息**，并让用户清楚它的好处。

11.7　渐进式Web应用的设计

由于我们的应用已经脱离了传统 Web 的界限，因此其设计也需要进行调整。

从应用在主屏幕上的图标开始，然后针对每种媒介的限制调整设计（例如，全屏渐进式 Web 应用没有地址栏和回退按钮，在网站模式下改变屏幕方向等），最后，无论网络条件如何变化，都能对应用充满信心。

11.7.1　设计应该反映条件的变化

我们在口头上讨论了要反映网络条件的变化，但是这也可以通过视觉传达出来。

你的应用可以自动禁用或者隐藏离线不可用的按钮或者功能，甚至可以修改这些按钮，以帮助用户理解将要发生的情况（例如，将发送按钮修改为“稍后发送”按钮）。

考虑一个渐进式 Web 应用，当你访问这个应用时，有大量的动态内容是按需缓存的。对于在离线时访问应用的用户，可能只有某些内容是可用的。你如何将这一点通过视觉反映给用户？

一个很好的例子是 Housing.com，它是印度的一个顶级房地产平台。

当访问者离线访问 Housing.com 时，那些没有缓存的列表和城市会是灰色的、不能选中的，而那些已经缓存的内容会正常显示。这个区别比较微妙，但是非常清晰（见图 11-4）。

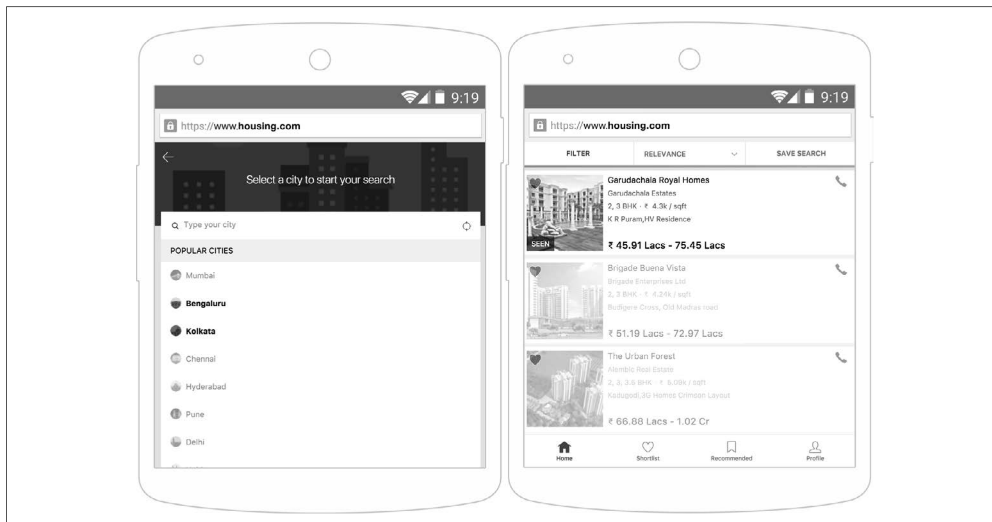


图 11-4：Housing.com 在离线时反映内容的可用性

连接性并不是你的设计能够反映的唯一变化条件。例如，你可以根据用户是否授予通知权限来修改你的界面，或者显示一个启用通知的按钮，或者显示控件来自定义需要显示哪些通知。

11.7.2　设计应该适应运行环境

现在，你的应用（和你的品牌）可以在浏览器之外、用户的主屏幕上显示，请确保它可以适应主屏幕，否则它就会像拇指一样突出。请不要在所有平台上重用已有的图标或 favicon。安卓主屏幕上的图标、Windows 10 瓦片、Safari 固定标签、MacBook Pro 的 Touch Bar 图标，它们之间的区别很大。要了解关于应用图标适配各种媒介的更多细节，请参阅 9.4 节。

11.7.3　设计应该适应每种媒介的特殊性

考虑一下，你的应用在全屏模式下运行，与在 Web 浏览器中浏览的效果是不一样的。缺少一个地址栏是否会影响品牌效果？你有没有精通技术的用户是通过复制粘贴网址来访问你的网站的？ HTTPS URL 旁边缺乏可视的安全指示是否会影响转化率？

解决这一问题的一种方法是添加额外的 UI 元素来提供类似的功能。通过使用 CSS 媒体查询，仅当显示模式设置为 `fullscreen` 或者 `standalone` 时，才显示这些 UI 元素：

```
@media all and (display-mode: fullscreen) {
  #back-button {
    display: block;
  }
}
```

11.7.4 设计应该向用户注入信心并通知用户

就像本章前面讨论的口头交流一样，你的设计也可以用来与用户交流，并灌注信任感和信心。

例如，当使用 WhatsApp 发送消息时，输入消息后就会在旁边出现一个灰色的复选标记。即使用户离线，也会发生这种情况。这有助于向用户保证，不管是离线还是在线，消息会被记录在应用中并尽快发送。如果你的应用也提供同样的可靠性，可以向用户保证，他们精心编写的消息在应用中是安全的。不要让用户感到疑惑。

11.7.5 设计应该帮助用户和企业实现目标

如果用户在飞行模式下，你的旅行软件也能工作吗？这是一个很强大的竞争优势。请确保你的用户知道。

让更多的用户注册推送通知，会有助于企业重新召回更多的用户吗？请确保与用户正确交流通知的工作方式，并将它的好处传达给用户。

11.8 负责安装提示

本章前面介绍了如何改善请求权限发送通知的用户体验。你可能想要改进的另一个提示是 Web 应用安装横条，具体来说就是它的显示时机。

安装提示的显示时机完全取决于浏览器。不幸的是，浏览器只能猜测显示安装提示的最佳时间，但是它不像你那样了解应用或者用户。

如果用户正在结账，而浏览器决定要显示安装提示，该怎么办呢？你真的想在那一刻分散用户的注意力吗？

幸运的是，浏览器给予了你一些控制权。

虽然你不能控制何时启动安装提示，但是你可以监听浏览器何时决定显示它，拦截该事件，并延迟到之后再显示（或者是取消显示）：

```
window.addEventListener("beforeinstallprompt", function(promptEvent) {
  promptEvent.preventDefault();
  setTimeout(function() {
    promptEvent.prompt();
  }, 2000);
});
```

这段代码监听了 `beforeinstallprompt` 事件，并在其触发时在事件上调用 `preventDefault()`，以阻止安装提示显示出来。随后，它会在等待两秒钟之后，使用事件的 `prompt()` 方法，手动启动安装提示的显示。

其中一个有趣的实现来自于 Flipkart，它是印度最大的电子商务零售商之一。Flipkart 首页头部的右侧包含了一个小加号图标（如图 11-5（左）所示）。这个图标会不时摇动，诱导用户尝试点击它。点击之后，Flipkart 就会显示一个蒙层，指引用户如何添加 Flipkart 的主

屏快捷方式（如图 11-5(中) 所示）。但是，如果在浏览器尝试显示应用安装横条之后点击这个链接——Flipkart 此时已经把引用拦截并存储了下来——就会显示应用的安装横条（如图 11-5（右）所示）。

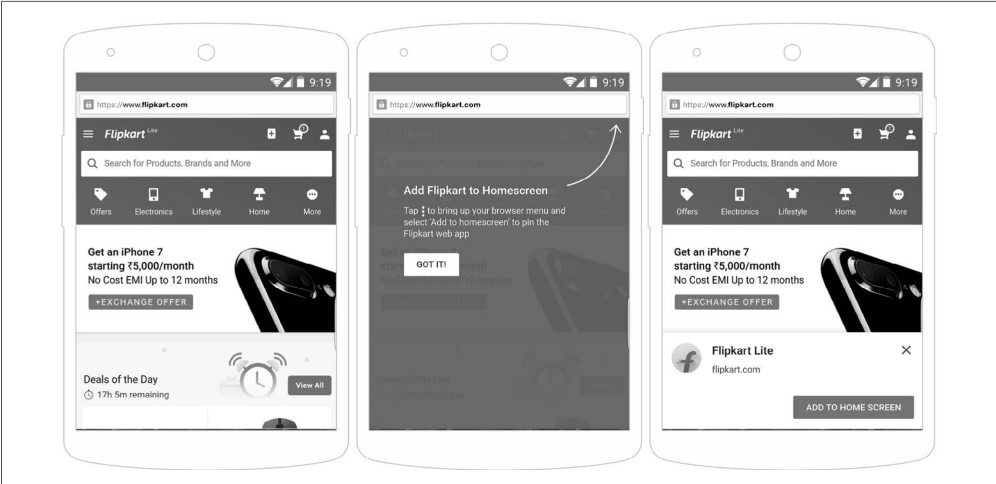


图 11-5：将 Flipkart 添加到主屏的体验

和推送通知一样，控制**时机**与**消息**是给用户提供美好体验的关键。

11.9　使用RAIL测量性能并实现高性能

一旦将应用放在主屏幕上，它和原生应用就是无法区分的。正如一句不太相关的俗语所说：能力越大，期望越大。当渐进式 Web 应用在用户主屏上获得了和其他原生应用同等的地位时，它最好能够像原生应用一样流畅地运行。

我们已经讨论过渐进式 Web 应用的许多强大的新功能，这些功能使得它们可以和原生应用一样强大。但同样重要的是使用的感受。

应用需要在用户使用时提供良好的感受。

要做到这一点，应用需要性能好、响应快、操作流畅。只有具备某些特点，才能让它感觉正确。这是一款能够迅速响应用户点击的应用与老式 Web 的体验之间的不同。在老式 Web 中，用户在点击之后要等待很长时间，怀疑点击是否注册了，有时候甚至要点击第二次来进行确认。当一个应用的感受良好时，所有这些疑虑都会消失。

要将这种对响应和性能的主观感受转化成可以测量并实现的东西，可以使用 RAIL 模型。[1]

RAIL 不是一种新技术或者新工具，它仅仅是一套指导方针，可以帮助我们理解什么可以使一款应用（原生、渐进式或者普通网站）的感受良好。和许多技术名词一样，RAIL 是一个缩写词，表示我们需要记住的准则：响应性、动画、空闲和加载。

注 1：RAIL 是由 Paul Irish 和 Paul Lewis 创造并定义的。

响应（Response）

当用户执行任何操作时，例如点击屏幕上的任何元素，我们希望能在 0.1 秒内做出响应。

你能在这个时间内显示用户请求的信息吗？如果能，这会非常棒！如果不行，你能否将屏幕内容过渡到用户请求的结果呢（即使它还没有包含实际的数据）？[2] 如果你做不到这一点，至少可以表明，用户的操作已被检测到，并且某件事情正在发生。最简单的方式就是显示一个加载指示器。

只要你能在 100 毫秒内对用户操作表现出某种响应，就会像是瞬间响应一样。努力给用户提供一种应用即时响应的感觉。不要让用户怀疑他是否点击了正确的东西，或者怀疑是否应该再点击一次。

记住牛顿第三定律：对于每一个操作，都应该有一个瞬间的响应。别跟牛顿作对。

动画（Animation）

要让动画在人眼中看起来流畅，它每秒至少需要更新 60 次。

要达到每秒 60 帧，意味着每 16.66（1000/60）毫秒要更新一次屏幕。由于浏览器还需要一些时间来将新的帧绘制到屏幕上，事实上每一帧只能获得 10 到 12 毫秒的时间。

请记住，当我们讨论动画时，指的是页面在用户滚动时的样子。当用户滚动时页面会卡顿，通常比起动画中任何其他延迟还要糟糕。

空闲（Idle）

将非必要的工作推迟到空闲时间。

“非必要”指的是任何不属于响应、动画或者加载的部分。用户是否在滚动一个无限长的条目列表？确保在加载和渲染下一批条目时，不会导致滚动“冻结”或者看起来卡顿。你需要下载并缓存一些资源，以供下次用户访问吗？请确保它不会减缓用户当前的访问速度。

加载（Load）

当用户执行一个操作，例如在网站上请求页面时，你的目标是在一秒之内显示操作的结果。

雄心壮志？也许吧。

可能吗？有了 service worker、CacheStorage、IndexedDB 和其他一些现代化工具包，答案是肯定的。

记住，你不需要在一秒之内加载整个应用，只需让用户感觉应用已经加载即可。有些时候，只需加载首屏的内容，并将其余部分延迟到空闲时间加载，就可以帮助你实现这一点（见图 11-6）。

注 2：你可以在图 11-6 中看到一个很好的示例。

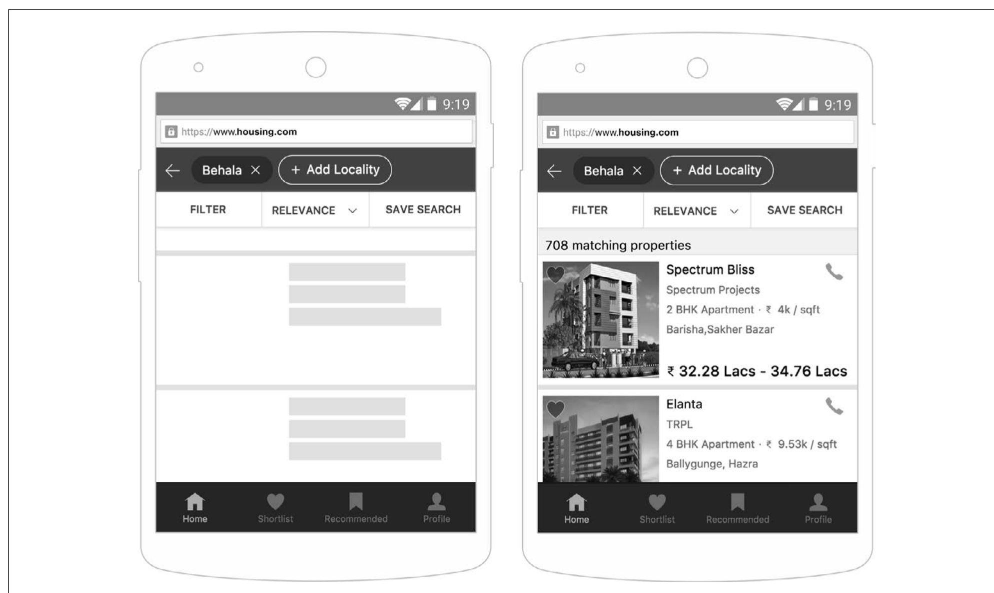


图 11-6：Housing.com 的搜索页面在结果可用之前就已经渲染出来

RAIL 的指导原则是：

- 在 100 毫秒或者更短时间内，显示对用户操作的某种**响应**；
- 确保每 16 秒（或者更短）绘制一次屏幕**动画**；
- 在页面**空闲**时执行工作，每次不超过 50 毫秒；
- 在 1000 毫秒内**加载**并显示用户请求的内容。

实现这些目标的具体细节超出了本书的讨论范围。你可以在 https:// pwabook.com/performancelinks 找到许多方便的资源来开启你的性能改进工作。

11.10　小结

渐进式 Web 应用为我们带来了新的 UX 挑战。但是处理得当时，渐进式 Web 应用也会带来很多提升用户体验和 Web 应用成功率的好机会。

有些内容可以直接添加到应用中，无须过多考虑对用户体验造成的影响，而其他内容则必须经过仔细思考才能添加。缓存静态资源并且始终从缓存中提供资源就是明摆着的事情，但是如果你添加推送通知的计划中仅包含引入一些代码，在用户一打开首页时就请求推送权限的话，你会收到一封来自管理人员的有趣邮件，其中会涉及漏斗、转化率和 KPI 的问题。

最后，用户体验比其他任何事情都重要。

当你使用本书中描述的任何技术时，第一步都是要停下来思考这些变化会如何影响用户体验。

第 12 章

渐进式Web应用的未来

让我们花点时间，看看目前完成的工作。

我们创建的渐进式 Web 应用在用户主屏上占有一席之地。它启动时会以全屏模式运行。无论用户在线还是离线，或者介于两者之间，它都是可用、快速且功能完备的。它甚至可以在用户离开网站之后，向其推送预订详情的变化。

除了不需要去应用商店安装应用之外，我们创造的内容与原生应用难以区分，甚至比原生应用更好。

如果你能想到任何优势或者特性是原生应用有而渐进式 Web 应用没有的，可能它已经存在了或者正在实现中了。

在最后一章，我们会快速介绍其中一些新技术，包括轻松接受支付的能力、简单用户登录的凭证管理、实时 3D 图像渲染、虚拟现实等。我们不会深入研究这些技术，其中一些还没有成型，但是我们会做简短的介绍，指引你去哪里可以学习更多内容。

12.1 使用Payment Request API接受支付请求

在线支付，尤其是在移动设备上，从来不是一件容易的事情——不仅因为开发者要尝试对接支付，而且用户要为填写长长的结账表单而挣扎。

虽然大多数在线购物已经可以在手机上完成，但是用户在台式机上完成交易的可能性比移动设备更大。原因很明显——在移动设备上填写长长的结账表单非常麻烦。另外，用户在线购物时的信任和安全问题也会造成影响，难怪移动设备的支付转化率比桌面设备要差。[1]

注 1：参见 2016 年 Monetate Ecommerce Quarterly Report,“How is everyone shopping, anyway?”。

应用商店为想要寻找应用收费或者订阅收费的原生应用开发者解决了一部分问题。他们为一键购买提供了可信的、共同的用户体验。而 Payment Request API 就能给 Web 带来了相同的体验。

它的目标就是消除冗长繁琐的结算表单。

对用户来说，Payment Request API 提供了标准化的支付界面，是属于设备原生的。它简化了定义支付方法和发货地址的过程，在结账付款时，用户可以轻松进行选择（见图 12-1）。

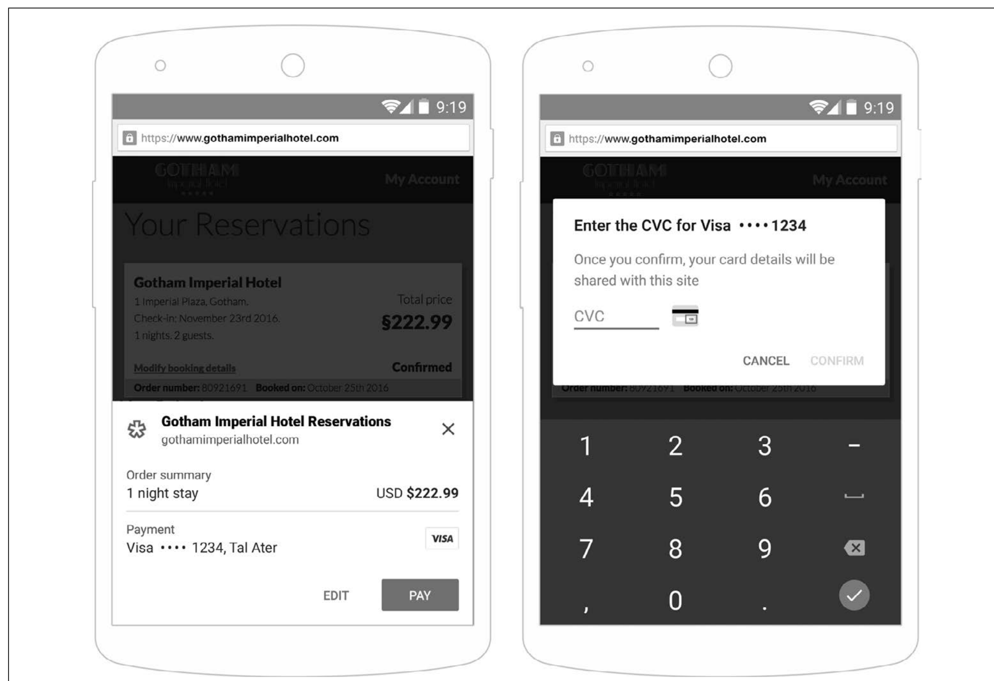


图 12-1：使用 Payment Request API 进行结算

对于开发者来说，Payment Request API 提供了一种大幅简化的方法，将支付整合到网站中：

```
var supportedPaymentMethods = [{
  supportedMethods: ["basic-card"],
  data: {
    supportedNetworks: ["visa", "mastercard", "amex", "discover", "diners"]
  }
}];

var orderDetails = {
  displayItems: [
    { label: "1 night stay", amount: { currency: "USD", value: "222.99" } },
    { label: "Holiday discount", amount: { currency: "USD", value: "-22.00" } }
  ],
  total: {
    label: "Total due", amount: { currency: "USD", value: "200.99" }
  }
```

```
};

var request = new PaymentRequest(supportedPaymentMethods, orderDetails);
request.show();
```

轻松接受应用和订阅的支付，可能是原生应用相比 Web 的最后一个优势了。随着 Payment Request API 提供一种更加优雅的方式来支付**任何东西**，Web 再次取得领先。

12.2 使用Credential Management API进行用户管理

对于大多数希望提供定制化用户体验的 Web 应用，用户需要登录，并在各个会话之间保持登录状态。这通常意味着用户需要注册、创建密码、记住密码，并经常使用它来登录到该服务。

但是，在移动设备上记住或者存储密码是一件麻烦事，以至于用户可能会简单地重用同一个密码来登录所有访问的站点，或者是在移动设备的网站上忽略登录（我是坚定的第二群人）。由于在移动端登录非常麻烦，很多用户往往仅仅因为这个原因就选择安装原生应用，而不是使用 Web 应用。

为了解决这个问题，一个称为 Credential Management API 的新标准就出来了。

Credential Management API 让开发者可以大大简化用户体验。用户一键即可登录，并且在会话过期时可以自动登录，甚至还可以记住曾经登录的联合账号（例如 Facebook Connect、谷歌账号等），并使用它们登录（见图 12-2）。凭证会本地存储在浏览器中，甚至可以在某些浏览器中在设备间同步。

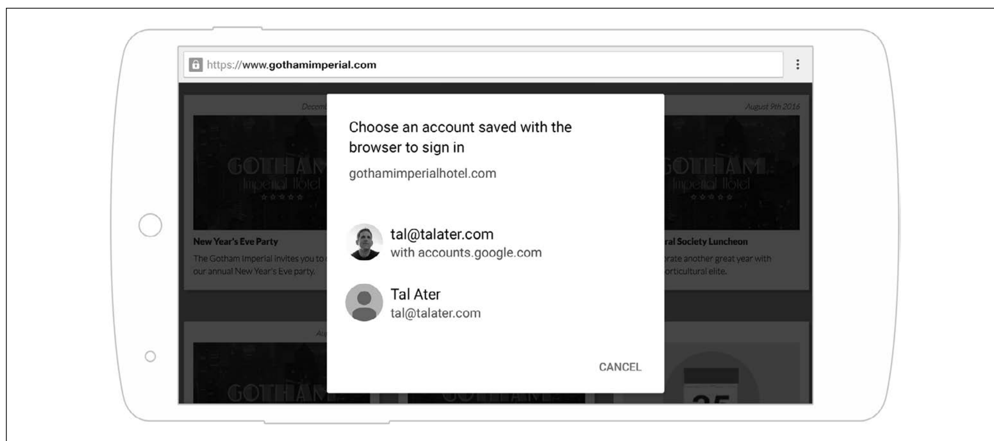


图 12-2：使用 Credential Management API 登录

如何使用 Credential Management API 可以根据你的网站需求而定。常见的工作流如下。

(1) 用户点击登录按钮，网站使用 Credential Management API 显示出本地账号选择器的界面。

(2) 如果用户登录成功，网站使用 Credential Management API 将凭证信息存储起来，以供将来使用。
(3) 如果用户过去已登录，并且会话过期，网站会重新登录该用户。

12.3 WebGL实时图像处理

在很长一段时间里，如果你想让游戏或者应用为用户带来强烈的视觉冲击，你的唯一选择就是转原生。DOM 根本不适合用来处理高级实时图形的处理要求。

如今，WebGL 在桌面和移动端浏览器中被广泛支持，让我们可以创建实时 GPU 加速的图形（见图 12-3）。

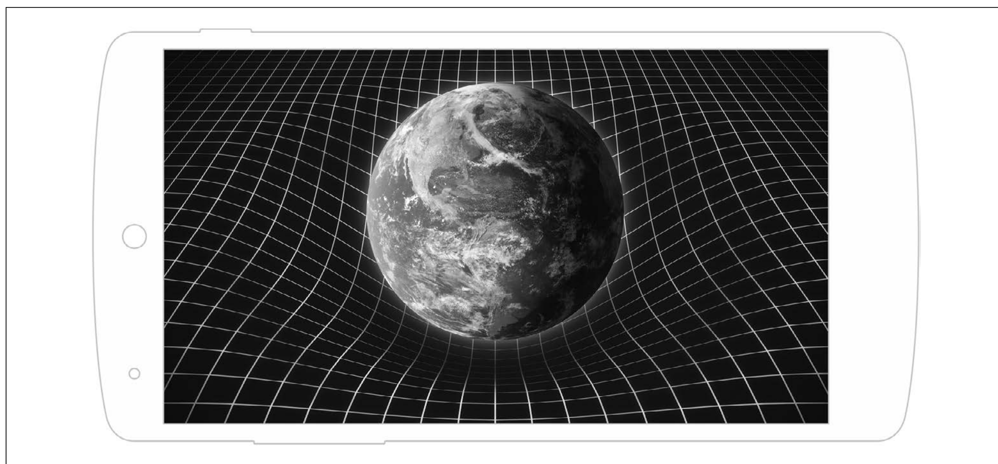


图 12-3：使用 WebGL 进行实时 3D 渲染

就像在 PC 和控制台上的原生应用或视频游戏那样，编写这样的高级图形，需要更多的数学、三角学专业知识，并不是我可以在大多数周一内能收集的。计算摄影机视角、光圈，以及在类似 C 语言的 GLSL 语言中编写自定义着色器，让 WebGL 入门的挑战变得相当大。

幸运的是，在游戏领域已经诞生了许多项目，简化了 JavaScript 开发者创建游戏和编写高级 2D、3D 图形的难度。

其中最流行的是 three.js，它可以轻松让你设置 3D 或者 2D 场景，添加摄影机、几何图形、材质等。

```
var scene = new THREE.Scene();
var camera = new THREE.PerspectiveCamera(75, 1.33, 0.1, 1000);
var renderer = new THREE.WebGLRenderer();
renderer.setSize(400, 300);
var geometry = new THREE.BoxGeometry(1, 1, 1);
var material = new THREE.MeshBasicMaterial({color: 0x00ff00});
var cube = new THREE.Mesh(geometry, material);
scene.add(cube);
camera.position.z = 5;
```

12.4 未来的语音识别API

很少有一种技术，既能提供令人惊讶的、流畅的、未来派的用户体验，又能为所有用户提高可用性和可访问性。但是，语音识别做到了。它为我们提供了一种全新的方式来与数字设备通信。这种方式既有未来感又很自然，同时也增强了可用性，并为我们的工作流程加速。这是一个全新的用户接口，是我们之前未曾用过的。

语音识别在过去是一个巨大的技术挑战。然而，如今的浏览器已经有了标准 API 来进行语音识别。这个 API 为 Web 开发者提供了轻松易用的接口，同时将背后的复杂技术细节交给了浏览器实现。

```
var recognition = new SpeechRecognition();
recognition.onresult = function(event) {
  console.log("User said: ", event.results[event.resultIndex][0]);
};
recognition.start();
```

这个易于实现的 API 对开发者来说是个好消息。不幸的是，也有坏消息。由于实际的语音识别需要消耗大量的计算资源，目前为止只有少数浏览器实现了这个 API。在本书编写时，语音识别可以在谷歌浏览器的桌面版和移动版上正常运行，在 Firefox 中也很快就可以使用。



对于尚未支持这个 API 的浏览器，要进行语音识别，可以使用 WebRTC 访问麦克风，然后使用微软必应语音 API 或者谷歌云语音 API 进行云端的语音识别。

要通过对开发者更加友好的方式添加语音识别到你的网站，请参阅 https://pwabook.com/annyang。

annyang 处理了浏览器之间的许多不一致性，并且尽可能简单地定义了语音命令：

```
annyang.addCommands({
  "What year is this?": function() {
    console.log("It is", new Date().getFullYear());
  }
});
annyang.start();
```

12.5 使用WebVR在浏览器中实现虚拟现实

谈到 VR（虚拟现实），Web 不甘落后，现在已经拥有自己的标准 API 来和 VR 设备进行交互，其中包括 Oculus Rift、HTC Vive、Google Cardboard 和 Samsung Gear VR。

WebVR 暴露了一个 JavaScript API，用于与设备显示器进行交互（包括单独渲染给每只眼睛）、通过 VR 输入设备（包括六个自由度设备）进行输入处理、读取用户的姿势等。

和 WebGL 一起使用，WebVR 可以让你呈现出复杂、令人信服的 VR 体验。

12.6 轻松共享应用

有两个新的 API 正在开发中，旨在让共享内容、链接和媒体变得更加容易，它们是 Web Share API 和 Web Share Target API。本质上，这两个 API 就像是同一个硬币的两面。

如今，当用户想要分享在网络上找到的某些内容时，他们有两种选择：要么使用内置在浏览器界面中的分享按钮（如果有的话），要么使用网站上的分享按钮——这是由网站所选择的、用来提供特定 Web 服务的按钮（例如点赞按钮、Twitter 转推按钮、Google +1 按钮等）。

Web Share API 允许网站添加一个通用的分享按钮，来触发设备的原生分享界面——这个界面会由用户安装在设备上的原生应用来填充：

```
navigator.share({title: "Gotham Imperial", url: window.location.href});
```

而 Web Share Target API 可以让 Web 应用进行注册，以便处理分享事件，就像原生应用那样。

通过 Web 应用清单，可以将应用注册为分享目标：

```
{
  "short_name": "ImperialApp",
  "name": "Gotham Imperial Hotel",
  "supports_share": true
}
```

一旦注册成功，应用会出现在原生分享界面中，就像其他原生应用一样（见图 12-4）。

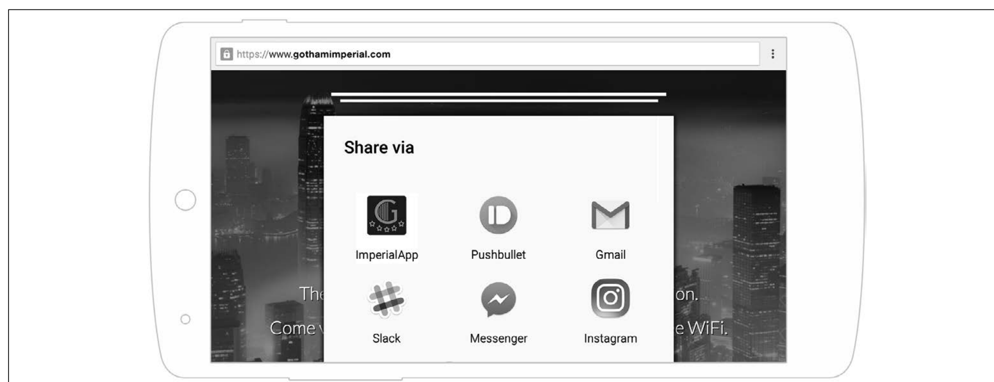


图 12-4：将 PWA 整合到原生分享界面中

此时，应用的 service worker 可以响应 share 事件，处理实际的分享：

```
navigator.actions.addEventListener("share", function (event) {
  var url = event.data.url;
  var title = event.data.title;
  var text = event.data.text;
  myShareFunction(url, title, text);
});
```

这两个 API 本质上是将社交分享大众化，并且创造了公平的竞争环境。这使得用户可以选择用什么应用来分享，并使得开发者可以将他们的 Web 应用用于社交分享。



在本书编写时，这两个 API 的细节仍在最后确定中。本节中所示的代码可能无法反映最终的 API。要了解更新的情况，请参考 https://pwabook.com/webshareapis。

12.7　流畅的媒体播放UI

如果你在开发一个渐进式 Web 应用来播放音频或者视频，那么你很幸运。新的 Media Session 标准允许你控制媒体如何显示到用户设备上，并让用户可以通过通知栏、锁定屏幕甚至是 Android Wear 这样的可穿戴式设备，进行媒体的播放控制。

即便没有定义任何内容，浏览器也可以在任何页面播放音视频时，在通知栏中显示通知。通知包括了浏览器基于播放媒体的页面或者应用，做出的对于标题的最佳猜测。

Media Session API 允许你设置媒体播放时要显示的元数据（并在下一个曲目开始播放时进行更新），包括标题、艺术家、专辑和作品。它还允许你设置用户点击播放、暂停、上一个、下一个按钮或者搜寻控件时，调用的事件处理器（见图 12-5）。

```
navigator.mediaSession.metadata = new MediaMetadata({
  title: "New Year's Mix",
  artist: "Gotham Imperial Hotel",
  album: "Gotham 2017",
  artwork: [{ src: "newyearmix.jpg" }]
});

navigator.mediaSession.setActionHandler("play", function() {});
navigator.mediaSession.setActionHandler("pause", function() {});
navigator.mediaSession.setActionHandler("seekbackward", function() {});
navigator.mediaSession.setActionHandler("seekforward", function() {});
navigator.mediaSession.setActionHandler("previoustrack", function() {});
navigator.mediaSession.setActionHandler("nexttrack", function() {});
```



图 12-5：渐进式 Web 应用的富媒体控制

使用 Media Session 标准以及本书中描述的技术，你就可以创建功能完备的媒体播放器，其中包括了完整的播放列表控件、在离线时播放音视频，甚至允许用户控制来自于已连接设备的回放。

12.8 下一个伟大时代

本书第 1 章第一节的标题是“Web 反击战”，真的再没有更加合适的方式来描述目前正在发生的转变了。

在 Web 初期，一切都是新的。人们蜂拥到台式机上，无限地获取信息，甚至第一次在网上购物。

帝国崛起，财富创造。

随后 iPhone 诞生，伴随而来的是移动互联网的新时代。但是那时候是 2007 年（IE7 的全盛时期），移动 Web 还没有准备好用户想要的丰富体验。因此，当苹果的应用商店在一年后上市时，原生应用很快抢走了风头。

帝国崛起，财富创造。

Web 一直是民主化获取信息和公平竞争的一个很好的均衡器。但是移动应用生态系统却变成了过于饱和的、管制的，并且严重倾斜向拥有雄厚资金的开发商。

但是，轮子总是在不断转动，现在 Web 又回到了前沿。

有了渐进式 Web 应用，用户不再需要一次安装几十个应用来访问他们所需的数据。如果用户选择不安装原生应用，他们就不再需要在体验上做出妥协。用户不再需要经历长长的安装漏斗，并给予每个应用无尽的权限。用户可以享受快速加载、离线访问、高可用、高可靠性，无论他们在哪儿。

有了渐进式 Web 应用，开发者不再需要对各个应用商店的规则或者用户体验进行妥协。他们不需要再花重金在应用商店有限的“前 10 应用”列表中保持竞争力。他们不需要再花费数周或者数月时间来开发，最终其应用却被应用商店因未知原因而拒绝。他们不需要再分别维护一个 iOS 应用、一个安卓应用和一个 Web 应用了。

渐进式 Web 应用最终让我们可以构建适合所有人的 Web 应用，无论用户的设备或者连接状态如何。它让我们可以构建丰富的体验，让用户可以长期使用，就像原生应用那样。

我一直在谈论渐进式 Web 应用让我们可以构建原生应用般的 Web 体验。但是，这仅仅是故事的开始。

从 Web 到移动 Web 的转变，以及后来从移动 Web 到移动应用的转变，让我们经历了以前从未想象过的体验。同样，转回到移动 Web 将带来我们无法想象到的惊奇新体验。

帝国崛起，财富创造。

这真是 Web 开发的大好时光。

附录A

service worker：采用ES2015的大好时机

ECMAScript 2015（又称为ES2015、ES6和ES6 Harmony）是ECMAScript语言规范（JavaScript实现的规范）在2015年的更新，也是自从2009年的ES5更新以来的第一个主要更新。

ES2015给ECMAScript（因此给JavaScript）添加了许多新的语言特性，其中包括箭头函数、常量、promise、类、模块、for/of循环、模板字符串等。

简单地说，它使得你在编写JavaScript时可以获得更加愉快的体验，并能帮助你编写出更优雅的代码。

不幸的是，对于想要使用ES2015编码的开发者来说，还有很多用户依然在使用旧式的不完全支持ES2015的浏览器。

这个问题可以通过使用Babel之类的工具，在编译时将ES2015代码转换为ES5旧代码来解决。这个过程会将你的代码中任何不兼容ES5的语法转译成兼容语法。不幸的是，很多开发人员对这个额外的构建步骤并不满意，或者选择不做，因此也就无法享受到这些新的语言特性了。

然而，service worker提供了一个很好的机会来开始使用ES2015。由于当前实现了serivce worker的所有浏览器中，同样实现了大部分的ES2015功能，因此你可以在service worker文件中安全地使用这些新功能，而不需要进行转译。

在我们的service worker中，我们已经用到了一些ES2015特性，包括promsie、`string.includes()`和`string.startsWith()`等。让我们看看可以使用ES2015来改进我们的service worker的其他方法。

A.1 模板字符串

模板字符串可以创建多行字符串，以及在字符串中包含变量，写法更加优雅。

和封装在双引号或者单引号中的字符串不同，模板字符串由反引号（`）包裹。通过使用反引号，就可以将多行字符串和占位符包裹起来。占位符由美元符号（$）和花括号（{}）表示，其中可以包含变量和表达式。

比较一下使用普通字符串和使用模板字符串拼凑出同一个字符串的方式。

使用普通字符串的多行字符串表达式：

```
var message =
"Nightly rate: " + rate + "\n"+
"Number of nights: " + nights + "\n"+
"Total price: " + (nights * rate);
```

使用模板字符串的多行字符串表达式：

```
var message =
`Nightly rate: ${rate}
Number of nights: ${nights}
Total price: ${(nights * rate)}`;
```

A.2 箭头函数

箭头函数提供了一种定义函数的简短语法，通常能够让代码更加优雅和富有表现力。

要注意，与传统的函数表达式不同，箭头函数会和周围的代码共享 `this`。

比较以下两种代码实现，它们响应来自 CacheStorage 的事件或者网络请求中的内容。

传统函数：

```
event.respondWith(
  caches
    .open("cache-v1")
    .then(function(cache) {
      return cache.match(event.request);
    })
    .then(function(response) {
      return response || fetch(event.request);
    })
  );
```

使用箭头函数实现同样的逻辑：

```
event.respondWith(
  caches
    .open("cache-v1")
    .then(cache => cache.match(event.request))
    .then(response => response || fetch(event.request))
);
```

A.3 对象解构

对象解构让你可以将对象中的特定值提取到不同变量中：

```
var reservationDetails = {nights: 3, rate: 20};
var {nights, rate} = reservationDetails;
console.log("Number of nights", nights);
console.log("Nightly rate", rate);
```

其中一种常见的用途是访问传递给函数的对象参数中的特定值。

比较下面两个示例可以看到在使用和不使用解构的情况下，如何访问传递给函数的对象属性。

传递对象作为一个参数：

```
var reservationDetails = {nights: 3, rate: 20};
var logMessage = (reservation) => console.log(
  `${reservation.nights} nights: ${reservation.nights * reservation.rate}`
);
logMessage(reservationDetails);
```

解构传入的对象：

```
var reservationDetails = {nights: 3, rate: 20};
var logMessage = ({nights, rate}) => console.log(
  `${nights} nights: ${nights * rate}`
);
logMessage(reservationDetails);
```

A.4 ES2015的更多内容

这些示例只是 ES2015 中引入的诸多语言新特性中的一小部分。

我鼓励你进一步探索 ES2015。通过代码，以及享受其中，你会受益匪浅。

附录B

全页间隙式广告

为了增加应用的安装量，许多网站会使用全页间隙式广告，将整个网站隐藏在一个移动应用的广告背后。

大量研究已经表明有多少用户讨厌这种广告。我甚至不会提供此类研究的链接，以免浪费你的时间。你可能已经猜到了答案。如果没有的话，欢迎访问“I Don’t Want Your F***ing App”的 Tumblr 页面。

但是让我们换个角度，问个不同的问题。全页间隙式广告是否有效呢？

2015 年，谷歌决定通过一项实验来回答这个问题。当谷歌发布全页间隙式广告的实验结果时，答案很明显。[1]

- 呈现全页间隙式广告之后，只有 9% 的用户点击了“获取应用”按钮（要记住，这只是安装漏斗的第一步）。
- 69% 的用户在间隙式广告打开之后就立即离开了页面。这些用户既没有去应用商店，也没有通过点击回到刚才的网站。

看到这些数字之后，谷歌决定再做一项实验。他们将间隙式广告替换成小型的、不显眼的应用横条广告，然后观察对产品实际使用的影响。结果令人惊讶：

- 移动网站的单日活跃用户增长了 17%；
- 原生应用的安装比例仅仅下降了 2%。

基于这项实验和其他实验的结果，谷歌决定放弃全页间隙式广告。2015 年 4 月，谷歌宣布，使用全页间隙式广告来推广原生应用的网站将不会得到排名的提升，其他移动端友好的网站则会提升。实质上，这意味着使用全页间隙式广告会导致网站的搜索结果受到影响。2016 年 8 月，谷歌采取了进一步的措施，对所有其他形式的间隙式弹出窗口进行了额外的排名惩罚。

注 1：参见谷歌网站管理员中心博文“Google+: A case study on App Download Interstitials”，发表于 2015 年 7 月 23 日。

附录C

CORS与NO-CORS

当网站对一个不同的源发起资源请求的时候，这个请求就被称为**跨源请求**（COR，cross-origin request）。例如，当地址为 https://www.gothamimperial.com/ 的页面尝试从 https://maxcdn.bootstrapcdn.com/ 加载样式，或者从 https://www.google-analytics.com/ 加载分析代码的时候，就属于这种情况。

出于安全原因，浏览器允许页面从不同的源**嵌入**资源，但是不允许脚本从另一个源**读取**资源内容。这就是所谓的**同源政策**。嵌入资源（例如当哥谭帝国酒店使用 `<link>` 标签从 CDN 加载样式表）是允许的，但是通过发起 Ajax 请求，从不同域的 JSON 文件中读取内容，就会被阻止。

开发者通常会通过嵌入资源代替直接访问（例如通过 JSONP）来绕过这些限制，但是这些只是在部分情况下能够使用的解决方案，而且这样做会将浏览器尝试解决的安全问题（主要是跨站脚本攻击）重新暴露给用户。

显然，需要更好的解决方案。

跨源资源共享（CORS，cross-origin resource sharing）是一个新的（不到十年）W3C 标准，用来定义服务器和浏览器之间的这些相互操作。浏览器构造请求以及服务器响应请求，共同决定了这个请求会被如何处理。举个例子，脚本可以配置一个不同源的请求。但是，要想请求成功，服务器还需要进行配置并响应跨源请求。服务器甚至可以配置为只接受来自于某些来源的请求（例如，www.pwabookcdn.com 可以配置成只响应来源于 www.pwabook.com 的跨源请求）。

在脚本中创建新请求时，你可以将模式设置成以下值中的一个。

`cors`

允许跨源请求。这是新请求的默认值。

`no-cors`

`no-cors` 这个命名有点让人混淆，实际上它是允许跨源请求的，但是这些 `no-cors` 请求受到的限制会比 `cors` 请求多。这些请求方法只能是 HEAD、GET 或者 POST 中的一个。如果 service worker 拦截了这个请求，只能够修改头部信息的一个有限集合。最终，JavaScript 代码不能够访问响应的属性。

`same-origin`

完全不允许跨源请求。

在第 5 章，我们必须在向服务器请求脚本的时候，设置只接受 `no-cors` 请求。如果没有将 https://maps.googleapis.com 的请求配置成 `no-cors` 模式，请求就会被服务器拒绝。通过只允许 `no-cors` 请求，服务器才能确保第三方站点能够读取数据，但是同时限制了其修改请求的能力。

```
if (requestURL.href === googleMapsAPIJS) {
  event.respondWith(
    fetch(
      googleMapsAPIJS+"&"+Date.now(),
      { mode: "no-cors", cache: "no-store" }
    ).catch(function() {
      return caches.match("/js/offline-map.js");
    })
  );
}
```

关于作者

塔勒·爱特尔（**Tal Ater**）是一名拥有 20 多年经验的开发者、顾问和企业家。他的经验涵盖了客户端、服务端、产品开发，以及研发和产品部门的管理。他非常热爱并参与了开源社区的工作。他的开源贡献包括广受欢迎的 service worker 库和语音识别库，每天有数百万人使用。他在 Web 开发、产品开发、安全和开源方面广泛著述，还发表过大量演讲。他的作品和研究在《福布斯》《纽约时报》和英国广播公司等媒体上广泛传播，让他的母亲感到非常骄傲。

关于封面

本书封面上的动物是戴胜（hoopoe，学名 Upupa epops）。它的名字是一个象声词，模拟了它的叫喊声。它原产于欧洲、亚洲、北非、撒哈拉以南非洲、马达加斯加的草地、稀树草原和林地。

戴胜是一种五颜六色的鸟，以其黄褐色头顶的独特橙色羽冠以及一对优雅的斑马条纹翅膀而著称。这些年来，它的外表为其赢得了许多仰慕者。在整个历史上，它拥有着崇高的地位。在古埃及王国的墙上和墓碑上，印有戴胜的描画，这个图像符号表明这个孩子是他父亲的继承人和继任者。在波斯，它是美德的象征。在欧洲，它代表了窃贼。在斯堪的纳维亚，它预示着战争。2008 年，戴胜被选为以色列的国鸟。

戴胜生活在洞穴中，会在树干、悬崖、墙壁等空间中寻找洞穴。戴胜在一个繁殖期内实行一夫一妻制。雌性孵卵，雄性喂养雌性。为了躲避捕食者，雌性会分泌出一种稠密的棕色液体，闻起来就像腐烂的肉。它还会把羽毛和蛋包裹在这些腐烂粘性物里，让这个窝非常令人讨厌。

戴胜主要以昆虫为食，如捕食性飞蛾的蛹，这是一种破坏性的森林害虫。由于这个原因，戴胜这个物种在许多国家得到了法律保护。

O’Reilly 封面上的许多动物都已经濒临灭绝，然而每一种动物对于地球都非常重要。要想知道如何为保护它们贡献你的一份力量，请到 animals.oreilly.com 进一步了解。

封面图片来自 *Meyers Kleines Lexicon* 一书。



微信连接



回复“Web开发”查看相关书单



微博连接

关注@图灵教育 每日分享IT好书



QQ连接

图灵读者官方群I：218139230
图灵读者官方群II：164939616

图灵社区
iTuring.cn

在线出版，电子书，《码农》杂志，图灵访谈

O'REILLY®

PWA开发实战

渐进式Web应用（PWA）综合了原生应用的优势以及Web的新功能和优点，同时规避了原生应用的问题，能为用户提供全新体验，是构建快速、可靠网站的利器。

本书通过将一个虚构的简单网站逐步改造成先进的PWA，帮助读者学习如何利用曾经专属于原生应用的特性来开发Web应用，使之能够快速加载、推送通知、离线访问、拥有更多控制权。

- **理解service worker的工作原理，并利用它创建在任何网络状态下都能瞬间启动的网站**
- **创建像原生应用一样可从手机主屏幕启动的全屏Web应用**
- **通过推送消息召回用户**
- **拥抱离线优先，构建可优雅地处理网络连接丢失的Web应用**
- **探索PWA给用户体验带来的新机遇与新挑战**

塔勒·爱特尔（Tal Ater），DAV Foundation联合创始人兼CTO，W3C汽车工作组特邀专家，拥有20多年经验的开发者、顾问和企业家。在Web开发、产品开发、安全和开源方面均有深刻见解和研究，相关著述在《福布斯》《纽约时报》和BBC等媒体上广泛传播。

“这本PWA实践指南简明、清晰，充分展示了PWA如何利用一系列新的标准化浏览器技术来实现原生应用的可靠性与能力。”

——**Andreas Bovens**
Mozilla产品负责人

封面设计：Karen Montgomery 张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机／Web开发

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版



ISBN 978-7-115-50200-1
定价：79.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks